TU WIEN Informatics

Bayesian inference for inverse software performance problems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Florian Fischer, BSc
Matrikelnummer 01529081

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Wien, 4. Mai 2023

Florian Fischer   Jürgen Cito

Informatics

Bayesian inference for inverse software performance problems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Florian Fischer, BSc
Registration Number 01529081

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Vienna, 4th May, 2023

Florian Fischer   Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Florian Fischer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Mai 2023

Florian Fischer

Danksagung

In erster Linie möchte ich meinem Betreuer, Jürgen Cito, dafür danken, mir die Themen “probabilistisches Programmieren” und “Bayes’sche Inferenz” näher zu bringen. Dadurch wurde definitiv mein Interesse an diesen Themen und am Thema dieser Diplomarbeit geweckt. Weiters möchte ich ihm für die Richtungsweisung im Bezug auf die dieser Diplomarbeit zugrundeliegenden Arbeit und auf den schriftlichen Teil dieser Diplomarbeit danken.

Als nächstes bedanke ich mich bei meiner Familie, meinen Freund*innen und meiner Partnerin für die geduldige Unterstützung und die wiederholten Fragen nach dem Status dieser Arbeit, einhergehend mit dem Aufzeigen von “sich gut anbietenden Zeiten”, um mein Masterstudium abzuschließen. Obwohl diese “sich gut anbietenden Zeiten” erstaunlich häufig vorkamen, war es dennoch ein guter Weg zur Motivation.

Ein besonderer Dank geht auch an Andreas Lackner, der sichergestellt hat, immer als Erstes nach dieser Diplomarbeit zu fragen, wenn wir uns trafen.

Acknowledgements

At first, I would like to thank my advisor, Jürgen Cito, firstly for introducing me to the topic of probabilistic programming and Bayesian inference, this certainly sparked my interest in these topics and this thesis's questions. Secondly, I would like to thank him for the guidance provided on the underlying work of this thesis as well as on the written part of it.

Further, I would like to thank my family, friends and partner for their long-term support and repeated check-ins in regards of the status of this thesis, while simultaneously pointing out good periods of time to finish my studies in. These "good" periods of time seemed to occur surprisingly often, but nonetheless posed a good motivational tool.

Special thanks also go out to Andreas Lackner, who made sure that asking for the status of this thesis was almost always the first thing he did whenever we met.

Kurzfassung

Performance, Laufzeiten und Antwortzeiten sind stetig verbesserbare Aspekte von komplexen Computersystemen, die nicht nur theoretisch, sondern auch praktisch große Relevanz zeigen. Durch Methoden wie Tracking, Tracing und Messen von Antwortzeiten lassen sich Probleme der Performance durchaus rasch bis zur auslösenden Komponente verfolgen, jedoch sind genau diese Methoden in der Praxis oft nicht anwendbar, da die zusätzliche Rechenbelastung oder die zusätzlichen Kosten für das Speichern der gesammelten Daten nicht in jedem Szenario tragbar sind. In solchen Fällen müssen Messungen oft auf das Überwachen der Antwortzeiten des gesamten Systems beschränkt werden, was die Suche nach der auslösenden Komponente im Falle eines Performance-Einbruchs deutlich erschwert.

Eine mögliche Hilfestellung für die oben genannte Suche könnte *Inferenz* bieten. Klassische Inferenz, wie durch "Machine Learning", benötigt jedoch große Mengen von Daten im Training der Modelle um verlässliche Ergebnisse zu liefern, was wiederum im oben genannten Szenario nicht tragbar ist. Eine mögliche Lösung dafür scheint *Bayes'sche Inferenz* zu sein, die es erlaubt, anhand von geringen Daten in Kombination mit bestehendem Wissen und Annahmen über das modellierte System, verlässliche Inferenz zu betreiben. Der Bayes'schen Inferenz liegt *Bayes' Theorem* zugrunde, was die Inferenz zu einer *probabilistischen* Inferenz macht. Um also ein Problem mit Bayes'scher Inferenz zu lösen, muss es erst probabilistisch modelliert werden.

In dieser Diplomarbeit gehen wir näher auf eine Möglichkeit des probabilistischen Modellierens von Computersystemen und deren Performance ein und evaluieren daraufhin Bayes'sche Inferenz an solchen Modellen, um einer möglichen Verwendung als Unterstützung in der Suche nach Komponenten nachzugehen, die beobachtete Performance-Probleme ausgelöst haben könnten. Als erster Schritt wird ein Prozess etabliert, um den Sourcecode von Systemen in einen Graphen zu übertragen, der die Performance des Systems und dessen Komponenten widerspiegelt. Dieser Graph wird daraufhin in ein probabilistisches Modell übersetzt, auf dem die Bayes'sche Inferenz durchgeführt werden kann. Im zweiten Schritt wird ein Framework beschrieben, das Vorlagen mit vordefiniertem Verhalten von häufig vorkommenden Komponenten bietet, um in der Modellierung zu unterstützen, und in weiterer Folge auch die Inferenz mit den resultierenden Modellen in der Programmiersprache *python* erlaubt. Der Modellierungsprozess sowie die Inferenz über das Framework werden daraufhin an drei simulierten Testsystemen und generierten Daten evaluiert, die repräsentativ für etwaige Strukturen von realen Systemen gewählt wurden. Weiters wird eine Case-Study an einem echten System durchgeführt, für welches im lokalen Netzwerk über Benchmarking, gemeinsam mit manuellem Auslösen von Performance-Last, Daten gesammelt wurden.

Die Ergebnisse der Evaluierung legen die, in gewissen Fällen, potenzielle Anwendbarkeit von Bayes'scher Inferenz für inverse Software Performance Probleme dar, zeigen jedoch auch die Empfindlichkeit der Inferenz bezüglich der Definition des Modells und der Konfiguration der Parameter auf. Desweiteren zeigt die Evaluierung eine Einschränkung dieser Art von Inferenz, da viele Systeme durch zu ähnliches Performance-Verhalten der Komponenten nicht mehr für die Inferenz der Performance geeignet sein können. Softwaresysteme und deren Performance können jedoch mit Hilfe des hier etablierten Graphen gut von Sourcecode zu probabilistischen Modellen übersetzt werden, was die Möglichkeit eröffnet diesen Prozess zu automatisieren. Für eine verlässliche Inferenz mit diesen probabilistischen Modellen wird jedoch Expertise zum tatsächlichen sowie möglichen Systemverhalten und sorgfältige Konfiguration der Inferenzparameter benötigt.

Abstract

Performance problems of complex computer systems and the resolution of such are an everlasting topic of improvement in theory as well as in practice. While tracking, tracing and measuring response times all throughout the system certainly are useful practices to track down causes of performance issues quickly, once occurred, they are often not feasible in practice due to the sometimes high overhead and cost introduced by such measures and the data gathered by them. In such cases it might only be feasible to monitor the overall performance of the system and then perform thorough analysis and in-depth search for the cause once an issue is observed.

In this thesis we explore the possible support of such analysis through inference. Classical inference (e.g. Machine Learning) though, relies on large amounts of data in the training process to produce reliable results, which again might not be feasible in the above-mentioned setting. A possible relief for this issue seems to be *Bayesian inference*, which allows for inference on little observed data by incorporating prior knowledge and assumptions on the subject of inference into the inference itself. Bayesian inference is based on *Bayes' theorem* and is, as such, a probabilistic method of inference. Thus, it poses the need of modelling the problem at hand probabilistically.

This thesis will look into the probabilistic modelling of computer systems and their performance and then evaluate Bayesian inference on such models to explore a possible support in solving inverse performance problems. We will do so by introducing a workflow to transform a system's source code to a graph reflecting the system's performance, which can then be translated into a probabilistic model. Further, a framework is proposed that provides templates with preconfigured behaviour for common system components, with which such models can then be easily implemented and inference be performed upon in *python*. The modelling workflow, as well as the use of inference through the proposed framework is then evaluated on three artificial example systems and generated data, chosen to reflect different possible real-life structures, and one case study of a real-life system that was benchmarked and manually brought to heavy-load behaviour.

The results of the evaluation show a potential applicability of Bayesian inference for inverse software performance problems in some cases, although bringing attention to firstly, the sensibility of the inference regarding the chosen parameters and the defined model and secondly, the fact that many systems might not be fit for this type of inference. Using the proposed graph, we are able to easily translate software systems and their

performance from source code to probabilistic model, which also opens up the possibility of automating the modelling process for inverse performance problems. Though for reliable inference using these models, expertise on the system's actual as well as possible performance behaviour and careful configuration of the inference itself is needed.

Contents

CHAPTER 1

Problem Definition

In this section we will shed some light onto the specific types of problems this thesis will be relevant to and define the boundaries of what will and will not be considered (Section 1.1), as well as present the thesis itself and the research questions behind it (Section 1.2).

This thesis aims to tackle inverse performance problems where only some degree of information is available about the system under investigation and only higher level observations can be made regarding its performance. An example of such system may be a web service with complex underlying components accessed via a given endpoint. Here one might want to investigate a drop in performance that is only observed on the outermost level, the response time of the endpoint, but may be caused by a single function or component that acted behind the endpoint and slowed down the overall performance. We neither aim to find "bugs" or faulty code causing performance issues, nor do we aim to pin down performance problems to certain parts of the system. We rather aim to guide engineers or anyone investigating such an inverse performance problem into the right direction, helping them find issues, bugs or simply inefficient architecture. This can be done with the results of *Bayesian inference* (Section 3.1), the posterior distributions. As shown in Section 3.1.2, they are formed from prior assumptions and reflect how these assumptions were adapted during inference for the modelled system to produce the observed data and thus help in root cause analysis by hinting to the right direction.

1.1 Problem space

We are looking to guide anyone investigating *performance* problems, so we mainly look at response times or run times of a system and its components and possible inference that can be done on them. What can be considered a *response time* or *run time*? For the sake of simplicity, we will use these terms interchangeably. Now, given an arbitrary computational component C, seen as a "blackbox" that can be invoked to perform some computing steps of arbitrary degree of complexity, we define the *response time* r_C of said component C to be the time it takes from invoking C to C finishing its computation.

Definition 1.1.1 (response time)**.** Let t_{start} be the time of invocation of C and t_{end} the time of completion of its computation, then $r_C = t_{end} - t_{start}$ is the response time of C with $r_C, t_{start}, t_{end} \in \mathbb{R}^+$ and $t_{end} > t_{start}$.

Further, we define a system S to be a set of components $S = \{C_1 \dots C_n\}$ with $n > 0$ which itself can be invoked. Although such invocations can be nicely described by a *call graph*, we will introduce the notion of a *performance dependency graph*, to better describe the performance composition of a system's invocation and the respective performance dependencies.

Definition 1.1.2 (Performance dependency graph)**.** Given a system $S = \{C_1 \dots C_n\}$, a *performance dependency graph* $G_S = \langle V, E \rangle$ is a directed, acyclic graph consisting of nodes $v \in V$ with $V \subset S$, representing each component $C_i \in S$ involved in the invocation at hand and edges $e \in E$ between said nodes representing a dependency in regards of performance. Such a performance dependency is given if component C_x invokes any number of other components C_y, $x \neq y$, causing the performance of component C_x to be a function of the invoked components' performances. Further, we choose one node $v_r \in V$ representing the invocation of the overall system/endpoint, which cannot be a dependency of any other component, this node is called the *root node*.

Now given a performance dependency graph as specified in Definition 1.1.2, we define the run time r_S of a system S to be again, the time it takes from invocation to completion of computation, but more specifically, to be a function of the root node's components' performances and some unknown "noise" $\sigma \in \mathbb{R}^+$ to incorporate possible computational overhead and uncertainty.

Definition 1.1.3 (run time of a system)**.** Given a system $S = \{C_1 \dots C_n\}$ and its performance dependency graph $G_S = \langle V, E \rangle$, we define the response time or run time $r_S : S^n \times \sigma$ of the system to be a function $r_S(r_{C_1}, \dots, r_{C_n}, \sigma)$, of the response times of the root node's ($v_r \in V$) dependencies and unknown noise $\sigma \in \mathbb{R}^+$.

To define our problem space, we will only consider problems of the following structure: Assume there is a system S consisting of n different components $C_1 \dots C_n$ (with $n > 0$) and each of those perform different tasks of different complexity and run time. Further assume, that the response time of the system as a whole, say r_S, as well as the response time of each function inside, $r_{C_1 \dots C_n}$, can be measured or estimated. Now, given a set of response times $R_S = \{r_{S,i} \mid 0 < i \leq m\}$ of a particular invocation and the respective performance dependency graph G_S of the system, where m is the number of response times given, what is the expected distribution of the response times r_{C_i} of each function of G_S?

In a more informal way, lets construct an example that acts as an illustration for the problem of interest. Assume you are the technical manager of a video encoding system. Your system takes raw video data as input via download from an external source, encodes it in a given standard and writes the output to a local file, providing a URL the user to access it. To stand out amongst competitors in the market, you try to optimize your encoding service regarding its speed and efficiency and monitor its general performance as a reference. Now imagine, that while monitoring the system, you observe a drop in performance. Of course you would like to know what caused that drop - did the download of the input file take longer due to network issues, was the encoding particularly hard or faulty, or was there something wrong with writing the output? One way to accomplish that would be through manual analysis - if you only would have kept track of the performance of the individual components of the system. Since that poses a large overhead in monitoring and storing all the data you decided to just monitor the overall performance of the system. Now the question arises: Can you infer some information about the system's internal components' performance just from the overall performance and general knowledge about the system? For a formal definition of this example, see Section 4.1.

1.2 Research questions

The main goal of this thesis is to answer the question if one can infer the distribution of the response times of a system's internal functions or components and argue what might have caused a performance drop, given just observations of the system acting as a whole and general knowledge about the system. Based off of this, we can formulate the following research questions:

1. How can we model inverse performance problems and what priors could be used?
2. Does Bayesian inference on these models provide reliable results?

In the course of this thesis we will try to answer these questions by analysis and practical experimentation. The next sections will introduce the general, formal approach to tackle the above-mentioned problem, along with a proposed solution approach in form of a framework (Chapter 4), the specific implementation of said framework in *python* using *Numpyro* (Chapter 5) and finally results and evaluation of different test-systems (Chapter 6).

CHAPTER 2

Related Work

Current research on performance problems in computer system revolves around prediction, which is done via different kinds of performance models spanning analytical methods, simulation, and machine learning [1]. These models describe the performance of a given system and thus allow to predict its behaviour under different conditions and situations. This is in general a great tool while planning and constructing systems and their requirements on hardware since one can test how a system will behave in predefined situations and identify causes of performance issues. But in a later phase of the system's lifecycle, when the system is already deployed and running for example, performance issues might still arise and one might need to know what caused it. With the prediction models one would need to reconstruct the exact situation in which the problem occurred, which is often not possible due to the system's complexity and due to missing data. For this, an inverse approach that allows reasoning starting from the observed behaviour and not from the given situation seems to be more fitting. As of writing this thesis, literature on this topic is quite limited.

One approach was presented by Cao et al. [2], where they use a specific type of queueing network to model the performance of a web server and then estimate its parameters by log-likelihood maximization. In a more general way, Mohammadi and Salehi-Rad [3] use Bayesian inference and prediction in the context of a queueing network. Their approach also allows for re-servicing any task inside the network if needed.

Even more general in the terms of performance modelling and solving performance problems, but still quite relevant for this thesis is a paper by Sutton and Jordan [4]. They show an approach to reason about queueing networks for internet services and discuss how to infer missing data for diagnosis of performance problems while considering different types of queues inside the network.

For a broader overview, a survey presented by Balsamo et al. [1] provides a good overview on techniques used in performance modelling and prediction, like queueing networks or simulation based solutions, and gives a full comparison of these different approaches.

Finally, there is also work tackling these performance questions and root-cause analysis in a non-Bayesian manner. Chow et al. [5] show an approach to automatically construct a model of internet service request execution by proposing a large amount of hypotheses and rejecting those that contradict the observed data. A more recent paper by Wu et al. [6] presents an approach that correlates occurring performance symptoms with the respective resource utilization to localize the root cause based on a graph modelling the anomaly propagation within a system's services. Since we are not aware of any work looking specifically into Bayesian inference used for inverse performance problems in general, we aim to tap into that area of research with this thesis.

CHAPTER 3

Preliminaries

This thesis presumes that the reader does have some prior knowledge in the field of computer science along with common programming languages and paradigms, as well as basic knowledge in statistics. With that in mind, this section will introduce the reader to some further concepts, which are not assumed to be prior-knowledge but still relevant to this thesis.

3.1 Bayesian inference

The reader might be familiar with *Bayes' Theorem*, nevertheless we will revisit it briefly since it essentially is the core of Bayesian inference.

3.1.1 Bayes' theorem

Although not published by himself, Thomas Bayes established an interesting approach on tackling inverse probability problems through propositions presented in [7]. These propositions essentially capture conditional probability and thus led to what we now know as *Bayes' Theorem*.

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)} \tag{3.1}$$

Formula 3.1 shows Bayes' Theorem in its commonly known form, where $P(A|B)$ is read as the probability of event A occurring, given that event B happened (analogously for $P(B|A)$) and $P(A)$, as well as $P(B)$, simply describe the probability of the respective event, A and B, occurring.

Now to understand the relevance of the theorem for this thesis, let us first look at what an inverse probability problem is. The more well known probabilistic problems are most likely classical probability problems, such as calculating the chances of winning the lottery or the probability of a specific series of die rolls or coin tosses to occur. For such problems we already know the underlying probabilities of each atomic event that influences the overall probability, so we can simply calculate the resulting chances. Now on the other hand, imagine a lottery where one would draw a ticket, which upon opening ends up being either a winning ticket or a losing ticket. Further, assume that we do not have any information on the total amount of tickets available or on the number of winning/losing tickets that are in the pool, we can only *observe* the outcome of the tickets drawn. The question we would like to ask now is, what is the ratio of winning and losing tickets, or in other words, whats the probability of winning this lottery? This is an example of an *inverse* probability problem, a problem where we do not have access to all the information needed to exactly compute the probabilities of interest.

Bayes tackled this lottery example in [7]. More specifically, he asked for the probability that the ratio of winning and losing tickets lies between two values. He goes on to describe the computation of the probability for the ratio lying between 9:1 and 11:1. The results show how such problems can be tackled: If one observes 10 losses and 1 win, the probability for the ratio being within the interval lies at 7.699%, if one observes 40 losses and 4 wins, the probability is about 15.25% and if one observes 100 losses and 10 wins, the probability is already at 25.06%. Finally if 10.000 losses and 1.000 wins are observed, the probability of the ratio lying between 9:1 and 11:1 comes out as being 97.42%. At the same time, Bayes gives the results for a slightly altered example where the guess is that the ratio is lower than 9:1. Here we can see an initial probability of 65.89% for 10 losses and 1 win and observe the results gradually *decrease* to already 44.11% for 100 losses and 10 wins.

These results nicely show that increasing observations of an event happening or not happening, directly influence the *confidence* one can have in an initial guess either positively or negatively, depending on the quality of the initial guess. This principle lies at the heart of *Bayesian inference*.

3.1.2 Inference

Even though we assume the concept of inference to be known to some degree by the reader, we will give a quick overview here to ensure a unified understanding since inference may be known from different areas of scientific focus. After that introduction, we will look further into Bayesian inference.

Inference in general, is a part of reasoning about a given context with the goal of gaining further information from already existing data. For example, deriving conclusions from given premises can already be seen as inference. More precisely, deriving particular conclusions from known/assumed-to-be-true premises is called *deduction* and often known from the field of logic, while inferring universal conclusions from particular premises is called *induction* and often known from formal proofs across different scientific fields. In the context of this thesis, the term inference will be understood in its general form of *deriving further information from given data.*

Classically, inference is performed by making use of data analysis and the properties of the given data to then draw conclusions and gain information about some unknown aspect of a given problem. So we are presented some amount of data and try to either directly infer conclusions by deduction and induction or try to make predictions by creating a model and mathematically training it to produce data that fits the properties of the already given data (e.g. Machine Learning). Now the Bayesian approach to inference on the other hand, clearly reflects some of the concepts previously described in Section 3.1.1. First we start by defining *prior beliefs* about the probabilistic event of interest in the form of *distributions* that describe the underlying parameters. These prior beliefs are of course subjective and thus depend on whoever is formulating them, but this is expected and wanted, since it is a core property that comes with the Bayesian mindset, as we will see in the comparison of the frequentist and Bayesian view in Section 3.1.3. So these prior beliefs will act as our "initial guess", similar to the guess in the lottery example above. As a next step, we update our beliefs with *observations*, i.e. data that has been gathered on the actual events that occurred so far. This updating is done using Bayes' theorem and thus, after updating our prior beliefs with the observed data, we end up with *posterior beliefs* (often simply called *the posterior*), again in the form of distributions. We can then use this posterior to argue about unknown, underlying parameters of the probabilistic event of interest or to make predictions about so far unseen data along with a measure of uncertainty about the prediction.

To further understand the step of updating prior beliefs with given observations, we will now look at a small, informal example. Assume a friend of yours challenges you to a small game to decide who has to pay for the next coffee. The game needs each of you to roll a six-sided die ten times and the one who lands a six more often wins, leaving the losing party to pay. The chances seem fair and you usually land lots of sixes when playing board games, so you believe in your luck and agree to play the game. Your friend hands you a die and you go first, scoring a six in 2 out of 10 rolls. Continuing with the game, you notice your friend is not using the same die as you were and you remember that they once told you about owning a "special" die that will roll a six $\frac{1}{4}$th the time. Obviously you get a bit suspicious of them using the biased die now, but since you are not entirely sure about it you decide to not call them out. Now, let A be the event of the die being biased, then we assume the probability $P(A) = 0.5$ to capture the uncertainty about it being biased as our prior belief. Further let B be the event of your friend rolling a six, then, since we know how biased the die might be, let $P(B|A) = \frac{1}{4}$ be the probability of rolling a six, given that the die is biased. We can describe the probability of your friend landing a six as the sum of the probabilities of the die being biased **and** rolling a six and the die being fair **and** rolling a six:

$$P(B) = P(A \text{ and } B) + P(\neg A \text{ and } B). \tag{3.2}$$

For that purpose, lets look at *joint probability.*

Definition 3.1.1 (Joint probability)**.** The probability of two events, A and B, occurring at the same time is called the joint probability $P(A \text{ and } B)$, such that

$$P(A \text{ and } B) = P(A) * P(B|A). \tag{3.3}$$

Now from the definition of joint probability, we arrive at the following equation for $P(B)$

$$P(B) = P(A) * P(B|A) + P(\neg A) * P(B|\neg A) \tag{3.4}$$

and since we know all those probabilities, we arrive at a final value of about 0.2083 for $P(B)$:

$$P(B) = 0.5 * \frac{1}{4} + 0.5 * \frac{1}{6} \approx 0.2083. \tag{3.5}$$

Coming back to the die-rolling game, we initially estimated the chance of the die being biased to 50%, but now you see your friend rolling their first six and you get a bit more suspicious, although it might still just be luck with a fair die. Let's look at that mathematically by updating our initial guess with the newly observed data. We want to know the probability of the die being biased given that your friend landed a six, which is $P(A|B)$, and this can be nicely described by Bayes' theorem:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}. \tag{3.6}$$

Filling in the known probabilities

$$\begin{aligned} P(A) &= 0.5, \\ P(B) &= 0.2083, \\ P(B|A) &= \frac{1}{4}, \end{aligned} \tag{3.7}$$

we arrive at an updated value, the posterior, of

$$P(A|B) = \frac{\frac{1}{4} * 0.5}{0.2083} \approx 0.60. \tag{3.8}$$

So after observing your friend rolling a six, your updated belief of the die being biased should now be at 60%. Let's see what happens if you observe your friend landing more sixes, in reality you would get more and more suspicious with each additional six observed. In general, the probability of rolling a six n times, given the die is biased can simply be described by

$$P(B^n|A) = \left(\frac{1}{4}\right)^n, \tag{3.9}$$

where B^n denotes event B occurring n times, since multiple rolls are independent of each other. Now we also need to compute the probability of your friend rolling a six n times in general, which then is

$$\begin{aligned} P(B^n) &= P(A) * P(B^n|A) + P(\neg A) * P(B^n|\neg A) \\ &= 0.5 * \left(\frac{1}{4}\right)^n + 0.5 * \left(\frac{1}{6}\right)^n. \end{aligned} \tag{3.10}$$

Lastly, we use the above equations to describe $P(A|B^n)$ by

$$P(A|B^n) = \frac{0.5 * \left(\frac{1}{4}\right)^n}{0.5 * \left(\frac{1}{4}\right)^n + 0.5 * \left(\frac{1}{6}\right)^n}. \tag{3.11}$$

Assuming you see your friend roll a six two times, how suspicious should you be about the die being biased? Using equation 3.11 to get a result of $P(A|B^2) \approx 0.6923$, we should be slightly more suspicious. Well, what about them landing a six 3, 4 or even 5 times? The chance for the die being biased would then be

$$\begin{aligned} P(A|B^3) &\approx 0.7714, \\ P(A|B^4) &\approx 0.8351, \\ P(A|B^5) &\approx 0.8836. \end{aligned} \tag{3.12}$$

Upon observing six rolls landing a six, the probability of the die being biased is already at almost 92%, meaning you can be pretty confident in calling your friend out, if you haven't done so earlier anyways.

This small example shows that, thanks to Bayes' theorem, we can update our prior beliefs about some event upon receiving new information and act in accordance to the resulting posterior belief. This is the essence of Bayesian inference.

3.1.3 Frequentist vs Bayesian

When looking into probability, one may encounter different ways to define the "probability" or "chance" of some event. Traditionally, a common way of thinking about probability is the *frequentist* view, which may feel familiar to some coming from studies around statistics and probability theory. In [8], Wassermann nicely summarizes the frequentist, as well as the Bayesian view on probability. In its essence, the frequentist view sees probabilities as relative frequencies. So saying that on a coin toss the chance for tail is 0.6 (i.e. 60%), would mean that for 60 out of 100 times the (seemingly unfair) coin is flipped, it lands tails up.

In contrast to that stands the *Bayesian* view on probability, which is also the way we think about probability in this thesis. When we talk about the probability of an event, we talk about our subjective beliefs about that event occurring. So for example, claiming that the probability of the reader drinking a coffee while reading this lies at 0.85 (i.e. 85%) simply means that I personally consider it to be rather likely that you are currently consuming a hot and caffeinated beverage to get through this text. Of course beliefs might change, especially upon gaining new information, but as we have seen above, this is anything but a problem since we can simply update our prior beliefs with new observations and end up with beliefs that then also reflect this new knowledge.

3.2 Probabilistic Programming

In this section, we will briefly look at *probabilistic programming*, the programming paradigm used in this thesis, then we continue with *Numpyro*, a library that allows for probabilistic programming in *python*, and conclude with a brief explanation of the *Markov Chain Monte Carlo* (MCMC) algorithm.

The various programming paradigms usually differ in their core concepts. For example, while in object-oriented programming objects are first-class citizens, in functional programming its functions and in logic programming its facts and rules which we use to form our programs. Similarly, in probabilistic programming we use probabilistic models and distributions to define and reflect real-world scenarios. While in object-oriented programming a variable might represent an object, in probabilistic programming this notion extends to probabilistic concepts, and variables represent for example distributions. The major advantages of this are firstly, that we can use it **alongside** traditional programming and secondly, that it allows for **branching** in probabilistic inference. Thus, we can incorporate uncertainty into various computations like simulations or inference to produce predictions and analyses more close to real events that have a multitude of factors influencing their outcome and, as done in this thesis, perform inference that requires branching depending on probabilistic variables.

3.2.1 Numpyro

One of the many possibilities of doing probabilistic programming is via the language *python*. Even then, there are multiple frameworks and libraries one can use to allow for easy implementation of probabilistic concepts, like *PyroPPL* [9], *PyMC* [10] and *Stan* [11]. For this thesis, we first explored inference using *PyroPPL* along with MCMC for inference and compared it to *Stochastic Variational Inference (SVI)*, since that seemed to provide faster performance. After further testing and looking into *PyroPPL* we encountered *Numpyro*, a lightweight alternative to *PyroPPL* backed by the *numpy* [12] package, which, in our case, provides a significant speed-up when performing inference via MCMC compared to *PyroPPL*. Since MCMC inference also seemed to be a better fit for modelling performance problems, we settled with *Numpyro* (version 0.11.0). We use it to define our model probabilistically with distributions from the package and to perform the Bayesian inference by running the MCMC algorithm also provided by the package. The exact usage and syntax of *Numpyro* can be seen in Chapter 5. Lastly, a small example showcasing the advantage of **branching** using *numpyro* can be seen in Source code 3.1. It showcases a small game where the prize-pool depends on a random decision made beforehand and where the prize itself is then sampled from the pool.

3.2.2 Markov Chain Monte Carlo (MCMC)

The term *Markov Chain Monte Carlo* is composed of two separate, well-known concepts: *Markov Chain* and *Monte Carlo*. Let us briefly explore both concepts separately.

```python
import numpyro
import numpyro.distributions as dist
from jax import random

with numpyro.handlers.seed(rng_seed=random.PRNGKey(1234)):
    go_left = numpyro.sample('go_left', dist.Bernoulli(0.5))
    if go_left:
        print("You went left")
        prize = numpyro.sample('prize_left', dist.Bernoulli(10.0, 100.0))
    else:
        print("You went right")
        prize = numpyro.sample('prize_left', dist.Uniform(1.0, 10.0))

    print("You won:", prize)
```

Source code 3.1: Simple branching in *numpyro*

Monte Carlo describes a group of randomized algorithms, that, as a trade-off for performance, sacrifice correctness of their results within a certain probability. As summarized by [13] these types of algorithms are used to model and simulate events observed in the real world, which come with an inherent randomness and can help in optimization problems or simply help generate samples from given distributions. A *Markov Chain* on the other hand, is the concept of generating samples based on each other, with the specialty that each sample only depends on the one previous sample. In other words, the generation of a sample within a *Markov Chain* depends solely on the previously generated sample and is not influenced by any of the samples before that.

Now the combination of those concepts, *Markov Chain Monte Carlo*, allows for sampling from distributions without knowing all properties of a distribution, as explained by [14]. Meaning through MCMC we can estimate posterior distributions without perfect knowledge of the priors, which is firstly often difficult to achieve analytically and secondly exactly what we are trying to achieve with Bayesian inference in this thesis.

CHAPTER 4

An Inverse Model for Software Performance Analysis

In this section, we will introduce the general approach that this thesis proposes for solving performance problems as described in Chapter 1. To do so, we will use an illustrative example which we will first formalise, then define the relevant parameters to finally model it and perform inference on it.

Take into consideration the example we introduced in Section 1.1. We will use the informal description of the problem scenario as a base for this example.

4.1 Formalisation

Let us define the given problem formally. In the considered scenario, we can easily encapsulate the system's structure using three inner components, each performing one of the key tasks of the system: downloading the asset, encoding the video and writing the output. Let C_{load} be the component responsible for retrieving the raw video input, C_{encode} be the component performing the encoding, C_{write} be the component writing the output file. Further, let $S = \{C_{load}, C_{encode}, C_{write}\}$ be a representation of the system of interest. Note, that we will proceed with a shorthand name for the components $C_{load}, C_{encode}, C_{write}$, respectively using *load*, *encode* and *write* to keep further definitions more readable. Now, as defined in Section 1.1, the response time r_S of the system can be interpreted as

$$r_S(r_{load}, r_{encode}, r_{write}, \sigma), \tag{4.1}$$

with $\sigma \in \mathbb{R}$ and r_S being a function of the run times of the components of S and the noise σ describing how the overall system's run time is composed. Now, looking at the specific run times $r_{load}, r_{encode}, r_{write}$, this is where the model becomes probabilistic. We define these run times to be probabilistic variables, each coming from a distribution specific to the respective component of the system. Lets simply assume a Normal distribution for each component, with $\mu_{load}, \mu_{encode}, \mu_{write}$ being the average expected run time of each component and $\sigma_{load}, \sigma_{encode}, \sigma_{write}$ the, in the context of the system, plausible variances. We denote that by

$$\begin{aligned} r_{load} &\sim N(\mu_{load}, \sigma_{load}), \\ r_{encode} &\sim N(\mu_{encode}, \sigma_{encode}), \\ r_{write} &\sim N(\mu_{write}, \sigma_{write}). \end{aligned} \tag{4.2}$$

In this manner, we can also further define the function r_S, which describes the system's run time as a whole. For simplicity, we will assume that the system's run time can be described by just the sum of the components' run times, but with the influence of the aforementioned noise or rather *uncertainty*. In practice we aim to gain this information through expertise on the system's architecture and complexity. So for now we will define r_S to be:

$$r_S \sim N(r_{load} + r_{encode} + r_{write}, \sigma), \tag{4.3}$$

again assuming a *Normal* distribution for this example, with the previously defined noise σ as variance. Now, what do we aim to answer? Given the above information and a series of response times for this system, what do the actual distributions for the run times $r_{load}, r_{encode}, r_{write}$ that produced this observed data look like and can we infer which component most likely caused the observed performance issue?

4.2 Parameters

To get one step closer to answering the above-mentioned question, we need to define the *parameters* for the inference we are going to do. We can see these parameters as the "bolts and nuts" which we will use to tweak our prior assumptions, bringing them closer to their actual (i.e. posterior) shape and which will also reflect the result of the inference.

Continuing with our example, what we want to do is guide anyone investigating the performance problem into the right direction. In other words, we want to know for each component, what the chance is that it caused (or was part of) the performance problem. We want to derive the probability of each component being a performance bottleneck. For this we need to branch our model to consider the case of each component being a bottleneck. We will introduce a new parameter p_C reflecting the probability of component C being a bottleneck. Based upon this parameter, we will draw the response times of the respective component during inference from either one of two different distributions: one representing the (measurable) regular response behaviour and the other one reflecting the (expected) behaviour as a performance bottleneck. Since we want to maintain a measure of uncertainty about the results, we will also draw this parameter p from a given prior distribution. Now formally, we define parameters $p_{load}, p_{encode}, p_{write}$, the chance of each component being the bottleneck, to be:

$$\begin{aligned} p_{load} &\sim U(a_{p_{load}}, b_{p_{load}}), \\ p_{encode} &\sim U(a_{p_{encode}}, b_{p_{encode}}), \\ p_{write} &\sim U(a_{p_{write}}, b_{p_{write}}), \end{aligned} \tag{4.4}$$

where $a_{p_{load}}, a_{p_{encode}}, a_{p_{write}}$ and $b_{p_{load}}, b_{p_{encode}}, b_{p_{write}}$ are again, parameters of the inference, initialized with prior assumptions. Since in our example we do not have any prior information about the likeliness of each of those components being a bottleneck, we sample from a *Uniform* distribution and we will initialize all lower limits a to 0.0 and all upper limits b to 1.0 to reflect the initial uncertainty. This leaves us with $p_{load}, p_{encode}, p_{write}$ now being:

$$\begin{aligned} p_{load} &\sim U(0.0, 1.0), \\ p_{encode} &\sim U(0.0, 1.0), \\ p_{write} &\sim U(0.0, 1.0). \end{aligned} \tag{4.5}$$

The drawn values for $p_{load}, p_{encode}, p_{write}$ will then be used during modelling to branch either towards drawing the component's response time from the distribution representing regular behaviour or drawing from the distribution representing a bottleneck behaviour, which directly affects the result of r_S and thus acts as an essential tweaking point in fitting the model to the observed data. Since $a_{p_{load}}, a_{p_{encode}}, a_{p_{write}}$ and $b_{p_{load}}, b_{p_{encode}}, b_{p_{write}}$ are parameters of the inference, those initial values we decided upon above will be tweaked when running the inference, with the goal of fitting the drawn values to the observed values, effectively fitting the priors to the unknown posteriors. After the inference, the updated parameters will reflect the result of the inference and we can look at the distributions of $p_{load}, p_{encode}, p_{write}$ to gain information on the possibility of each component being a bottleneck along with the corresponding uncertainty regarding that statement.

4.3 Modelling

Now in this section, we will take a look at modelling the whole system while considering the formalisation and parameters defined in Section 4.1 and Section 4.2. First, we will establish a brief overview of what we have defined so far.

We are considering a system S, consisting of three components:

$$S = \{load, encode, write\}. \tag{4.6}$$

Then we specified the response time of the system to be a function r_s of the components' response times $r_{load}, r_{encode}, r_{write}$ and some estimated noise σ:

$$r_S(r_{load}, r_{encode}, r_{write}, \sigma) \sim N(r_{load} + r_{encode} + r_{write}, \sigma), \tag{4.7}$$

with r_S drawing from a Normal distribution with the sum of the components' response times as mean and the noise as variance. The response times of the components themselves are, for simplicity, also drawn from a Normal distribution:

$$\begin{aligned} r_{load} &\sim N(\mu_{load}, \sigma_{load}),\\ r_{encode} &\sim N(\mu_{encode}, \sigma_{encode}),\\ r_{write} &\sim N(\mu_{write}, \sigma_{write}), \end{aligned} \tag{4.8}$$

with $\mu_{load}, \mu_{encode}, \mu_{write}$ and $\sigma_{load}, \sigma_{encode}, \sigma_{write}$ reflecting prior beliefs on the components' behaviour. Next we have defined the parameters of the inference $p_{load}, p_{encode}, p_{write}$ as:

$$\begin{aligned} p_{load} &\sim U(a_{p_{load}}, b_{p_{load}}),\\ p_{encode} &\sim U(a_{p_{encode}}, b_{p_{encode}}),\\ p_{write} &\sim U(a_{p_{write}}, b_{p_{write}}), \end{aligned} \tag{4.9}$$

where $a_{p_{load}}, a_{p_{encode}}, a_{p_{write}}$ and $b_{p_{load}}, b_{p_{encode}}, b_{p_{write}}$ are again parameters and will reflect our prior beliefs on the probability of each component being a bottleneck.

Now to start modelling the system and its behaviour, we need to consider that the behaviour obviously changes in case a component actually is a bottleneck. So we will now introduce separate response times r_b for that, which initial values will later be estimates approximated with the help of prior knowledge on the components' internal structures:

$$\begin{aligned} r_{b_{load}} &\sim N(\mu_{b_{load}}, \sigma_{b_{load}}),\\ r_{b_{encode}} &\sim N(\mu_{b_{encode}}, \sigma_{b_{encode}}),\\ r_{b_{write}} &\sim N(\mu_{b_{write}}, \sigma_{b_{write}}), \end{aligned} \tag{4.10}$$

with $\mu_{b_{load}}, \mu_{b_{encode}}, \mu_{b_{write}}$ and $\sigma_{b_{load}}, \sigma_{b_{encode}}, \sigma_{b_{write}}$ again reflecting our prior beliefs, but this time on the components' bottleneck behaviour. Now to actually have the model represent a real system, we need to apply our prior beliefs to the above definitions. The core values we need to populate are σ as the noise for the overall system's response time, $\mu_{load}, \mu_{encode}, \mu_{write}$ and $\sigma_{load}, \sigma_{encode}, \sigma_{write}$ for the components' response times, as well as $\mu_{b_{load}}, \mu_{b_{encode}}, \mu_{b_{write}}$ and $\sigma_{b_{load}}, \sigma_{b_{encode}}, \sigma_{b_{write}}$ for the components' bottleneck behaviour. Assume that, from prior investigation and existing knowledge, we know component *load* has a mean response time of $520ms$ and might vary by about 15%. So we will set $\mu_{load} = 520$ and $\sigma_{load} = 78$. Similarly, we investigate the behaviour of *encode* and *write* and result with a mean response time of $1200ms$ for *encode* with 9% variation and a mean of $700ms$ with 8.7% variation for component *write*. To reflect that, we set $\mu_{encode} = 1200, \sigma_{encode} = 110$ and $\mu_{write} = 700, \sigma_{write} = 61$. We now arrive at definitions of $r_{load}, r_{encode}, r_{write}$ being:

$$\begin{aligned} r_{load} &\sim N(520, 78),\\ r_{encode} &\sim N(1200, 110),\\ r_{write} &\sim N(700, 61). \end{aligned} \tag{4.11}$$

In the same manner, we will define the bottleneck behaviour of the components with $\mu_{b_{load}} = 780$ and $\sigma_{b_{load}} = 80$, $\mu_{b_{encode}} = 2400, \sigma_{b_{encode}} = 150$ and $\mu_{b_{write}} = 1000, \sigma_{b_{write}} =$

100. So for $r_{b_{load}}, r_{b_{encode}}, r_{b_{write}}$ we are left with:

$$\begin{aligned} r_{b_{load}} &\sim N(780, 80), \\ r_{b_{encode}} &\sim N(2400, 150), \\ r_{b_{write}} &\sim N(1000, 100). \end{aligned} \tag{4.12}$$

Given the above-mentioned performance metrics and previous response times of the whole system, we estimate an additional noise of roughly $100ms$, so:

$$\sigma = 100. \tag{4.13}$$

Finally, we still have to apply our beliefs of each component being a bottleneck to $\mu_{b_{load}}, \mu_{b_{encode}}, \mu_{b_{write}}$ and $\sigma_{b_{load}}, \sigma_{b_{encode}}, \sigma_{b_{write}}$. We can do this with the help of historical data on the components' behaviour or by using the actual beliefs of human experts on the system. In this example we will assume we have access to neither of those, so we will initialize the a_p values to 0.0 and the b_p values to 1.0 to incorporate the indifference with a Uniform distribution across the whole range:

$$\begin{aligned} p_{load} &\sim U(0.0, 1.0), \\ p_{encode} &\sim U(0.0, 1.0), \\ p_{write} &\sim U(0.0, 1.0). \end{aligned} \tag{4.14}$$

With the above defined equations 4.6, 4.7 as well as the initialized parameters from equations 4.11, 4.12, 4.13 and 4.14, we have specified the probabilistic model for our example system. In the next section, we will look at the actual inference.

4.4 Inference

As described in Section 3.1.2, the Bayesian approach to inference is to formulate prior beliefs and then update those beliefs upon observation of new information. We formulated our prior beliefs in Section 4.3 in the form of probability distributions for the response times of the system's components and for the additional parameters we specified. So as a next step we need to gather information and update our beliefs accordingly. Since we have constructed an artificial example, we will also use artificial data. So assume that we observe the set D of response times, with $|D| = 100$ as seen in 4.15, that reflect a performance dip caused by one of the components.

$$D = \left\{ \begin{array}{cccccccccc} 2701.30 & 2623.47 & 2767.75 & 2924.82 & 2575.98 & 2514.98 & 2917.04 & 2630.05 & 2655.33 & 2775.37 \\ 2711.42 & 2667.82 & 2874.87 & 2529.96 & 2499.32 & 2787.52 & 2701.09 & 2442.00 & 2687.62 & 2798.60 \\ 2704.09 & 2807.58 & 2863.89 & 2925.59 & 2677.72 & 2702.99 & 2685.61 & 2445.80 & 2476.92 & 3026.57 \\ 2818.25 & 2566.54 & 2618.43 & 2645.32 & 2608.02 & 2721.08 & 2843.06 & 2691.85 & 2652.01 & 2577.47 \\ 2672.22 & 2810.54 & 2691.33 & 2384.59 & 2678.48 & 2672.20 & 2764.34 & 2411.14 & 2970.18 & 2774.16 \\ 2751.16 & 2798.44 & 2773.11 & 2618.21 & 2535.41 & 2610.33 & 2732.65 & 2785.11 & 2747.65 & 2654.94 \\ 2785.24 & 2633.07 & 2358.42 & 2922.41 & 2456.58 & 2753.29 & 2761.24 & 2719.73 & 2707.35 & 2830.13 \\ 2403.19 & 2799.95 & 2412.42 & 2787.04 & 2601.00 & 3026.41 & 2453.55 & 2616.57 & 2794.98 & 2690.81 \\ 2632.55 & 2685.23 & 2949.13 & 2541.15 & 2887.59 & 2836.84 & 2779.09 & 2844.90 & 2593.76 & 3068.61 \\ 2744.90 & 2657.41 & 2646.53 & 2810.41 & 2542.58 & 2878.71 & 2890.39 & 2728.12 & 2744.13 & 2672.38 \end{array} \right\} \tag{4.15}$$

We have generated this "observed" data by first sampling r_{load}, r_{encode} and $r_{b_{write}}$ from the above defined distributions to reflect a scenario where the *write* component behaved as a bottleneck, so

$$\begin{aligned} r_{load} &\sim N(520, 78), \\ r_{encode} &\sim N(1200, 110), \\ r_{b_{write}} &\sim N(1000, 100). \end{aligned} \tag{4.16}$$

As specified in the model, we then sampled the overall response time r_S from the sum of these response times along with a slight variance $\sigma = 100$:

$$r_S \sim N(r_{load} + r_{encode} + r_{b_{write}}, \sigma). \tag{4.17}$$

This composed sampling was performed 100 times to generate the data shown in 4.15. Now we can actually perform inference as we have seen in Section 3.1.2. For the sake of readability we will perform inference for one component fully, to demonstrate the process, and later show the results for each component for comparison. Initially, we are indifferent about which components might be bottlenecks, so we will use a probability of 0.5 for each:

$$\begin{aligned} p_{load} &= 0.5, \\ p_{encode} &= 0.5, \\ p_{write} &= 0.5. \end{aligned} \tag{4.18}$$

From now on though, we will only be looking at the output component *write*. First, let us recapitulate the goal of the inference. We want to know the probability of component *write* being slow, given the observed data. Let A_{write} be the event of component *write* being a bottleneck and B be the event of us observing a given value x as the system's response time r_S, $P(A_{write}|B)$ is the probability of component *write* being a bottleneck, given r_S was observed to be x (with a variance of $\sigma = 100$). We know $P(A_{write}|B)$ is defined as

$$P(A_{write}|B) = \frac{P(B|A_{write}) * P(A_{write})}{P(B)}. \tag{4.19}$$

So what remains to fill in are the probabilities $P(B|A_{write}), P(A_{write})$ and $P(B)$. We do have an initial assumption about the probability of A, so

$$P(A_{write}) = 0.5. \tag{4.20}$$

Further, we can describe $P(B)$ as the combined probability of A_{write} **and** B occurring together and the inverse of A_{write} occurring together with B, so

$$P(B) = P(A_{write} \text{ and } B) + P(\neg A_{write} \text{ and } B). \tag{4.21}$$

Which, with the definition of *joint probability* (see Definition 3.1.1), can be rewritten as

$$P(B) = 0.5 * P(B|A_{write}) + 0.5 * P(B|\neg A_{write}). \tag{4.22}$$

Now as a last step we need to find $P(B|A_{write})$, which is the probability of observing r_S as the value x (with variance $\sigma = 100$) given that component *write* is a bottleneck. Given our estimates for regular and bottleneck behaviour of the components, we can formulate a term to reflect the variance in the response time r_S while considering a component to be a bottleneck:

$$P(B|A_{write}) = 1 - \left|1 - \frac{\frac{obs}{min} + \frac{obs}{max}}{2}\right|. \tag{4.23}$$

Here we take the mean of the relative difference of the observed value compared to the minimum and maximum expected r_S considering the variances of the components' response times as well as of the system's response times. So since we are considering the case of component *write* possibly being a bottleneck, we have minimum and maximum expected values for r_S of $\mu_{load} - \sigma_{load} + \mu_{encode} - \sigma_{encode} + \mu_{b_{write}} - \sigma_{b_{write}} = 2432 - \sigma = 2332$ and $\mu_{load} + \sigma_{load} + \mu_{encode} + \sigma_{encode} + \mu_{b_{write}} + \sigma_{b_{write}} = 3008 + \sigma = 3108$. So considering the first observation in D we result in:

$$P(B|A_{write}) = 1 - \left|1 - \frac{\frac{2701.3}{2332} + \frac{2701.3}{3108}}{2}\right| = 0.9862. \tag{4.24}$$

Analogously for $P(B|\neg A_{write})$ we result in

$$P(B|\neg A_{write}) = 1 - \left|1 - \frac{\frac{2701.3}{2071} + \frac{2701.3}{2769}}{2}\right| = 0.8601. \tag{4.25}$$

Filling the above result into the equation for $P(B)$ leaves us with

$$P(B) = 0.5 * 0.9862 + 0.5 * 0.8601 = 0.9232 \tag{4.26}$$

and we can continue with our inference by using these results in computation of $P(A_{write}|B)$:

$$P(A_{write}|B) = \frac{0.9862 * 0.5}{0.9232} = 0.5341. \tag{4.27}$$

The result of $P(A_{write}|B) = 0.5341$ shows that after observing the first response time in D, we already see the probability of component *write* being a bottleneck as increasing. Updating the formula of $P(B|A_{write})$ to allow for more than one observed value, will help us consider the whole observed dataset D:

$$P(B^D|A_{write}) = \prod_{i=1}^{n} 1 - \left|1 - \frac{\frac{d_i}{min} + \frac{d_i}{max}}{2}\right|, \tag{4.28}$$

where $n = |D|$ and d_i represents the i-th element of D. Using this, we can easily calculate the value of $P(A_{write}|B^D)$ after observing all 100 of response times within D:

$$P(A_{write}|B) = 0.99997841. \tag{4.29}$$

As we can see, the result leads us to being quite certain that component *write* might have been a bottleneck while observing the data D. Figure 4.1 shows a plot of the intermediate results while the considered observed data increases and nicely shows how the confidence in component *write* being a bottleneck rises along with the observed evidence.

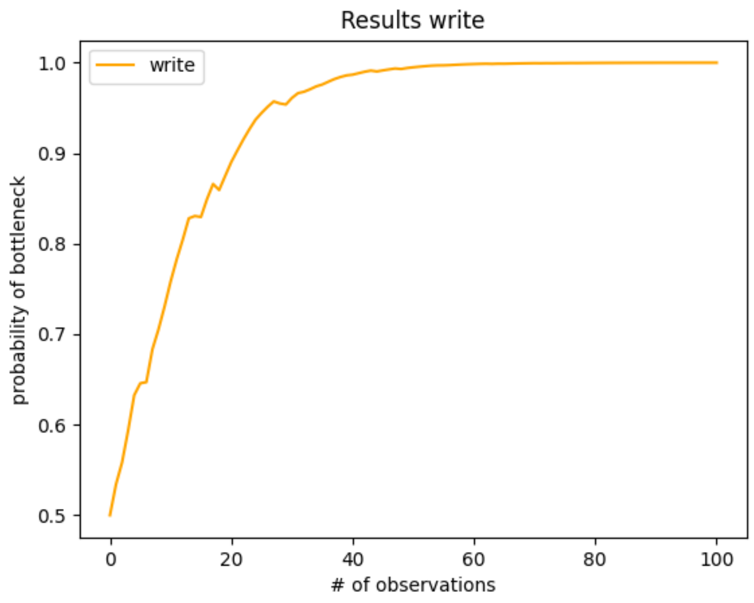


Figure 4.1: Intermediate results of inference on component *write* with increasing number of considered observations.

Lastly, as a comparison, we will look at the results of the inference on the other two components:

$$\begin{aligned} P(A_{load}|B) &= 0.99996476, \\ P(A_{encode}|B) &= 2.92578413 * 10^{-6}. \end{aligned} \tag{4.30}$$

Figures 4.2 and 4.3 show the plots of the intermediate results of inference for component *load* and *encode* respectively.

As we can see, the result for component *encode* clearly shows confidence in *encode* not being a bottleneck while observing data D, which is true in our artificial scenario. On the other hand the result for component *load* shows confidence in *load* being a bottleneck during observation, which we know is not true, but upon further inspection of the regular as well as bottleneck behaviour of the true bottleneck *write* and the hereby inferred bottleneck component *load* we can see why this result still is reasonable for the inference. Both components show a similar behaviour in the regular and bottleneck case. The difference in behaviour might be marginal, but we can still also see a difference in the results of components *load* and *write*, with $P(A_{write}|B) = 0.99997841$ showing slightly more confidence than $P(A_{load}|B) = 0.99996476$ although the difference is ever so slightly. If we plot the results for *load* and *write* next to each other, as we can see in Figure 4.4, the difference in the results gets a bit more clear, showing *load* resulting in values slightly lower than those for *write* throughout the process and especially dipping further down when observing values that lead against the prior assumption.

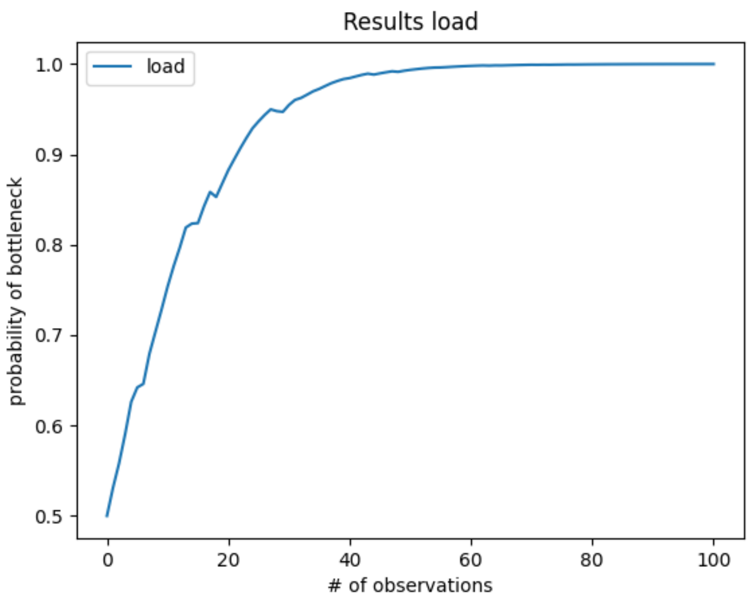


Figure 4.2: Intermediate results of inference on component *load* with increasing number of considered observations.

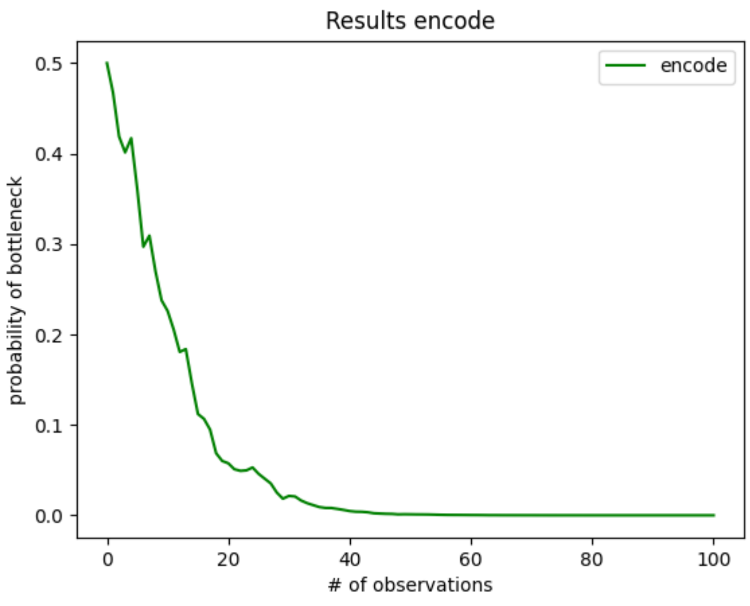


Figure 4.3: Intermediate results of inference on component *encode* with increasing number of considered observations.

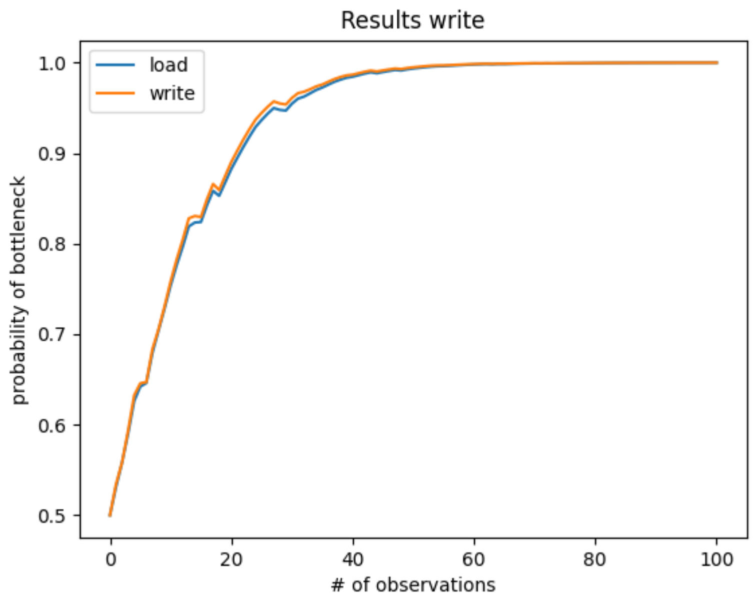


Figure 4.4: Intermediate results of inference on component *load* along with the results of inference on component *write*.

4.5 Framework

Now to make the use of Bayesian inference for inverse performance problems more accessible, we propose a framework [15] that aids in modelling the problem and performing the inference. In this section, we will first look at the concept of this framework and then showcase the modelling of an example system. Chapter 5 will then show the implementation and actual usage.

Figure 4.5 shows the conceptual usage of the framework with an example system as input to showcase a possible model structure. It takes as input the observed data (i.e. the observed response times of the system/endpoint) and the system/endpoint itself. The latter is defined by a tree-like structure with nodes representing the system components and the edges representing performance dependencies between them. Each node specifies their response time behaviour either by sampling from a distribution or as a function of their child-nodes' response times. The nodes themselves are defined by a name, their behaviour function and possibly a list of child nodes. Additionally each node maintains an `is_slow` flag representing the state of it being a bottleneck or not. One core part of the framework, besides the inference, is that it provides templates for common computer system's components, that exhibit predefined regular and bottleneck behaviour. These templates are based on performance behaviour observed in relevant literature and can be tweaked to allow for specific modelling. We provide these predefined node templates for common components such as *CPU*, *network*, *database*, *memory* and *queue*, which all

Component	Distribution	References
CPU	LogNormal	[16], [17]
Network	LogNormal, Pareto	[18]
Database	LogNormal	[19]
Memory	LogNormal, Weibull	[20], [21]
Queue	Exponential	[20]

Table 4.1: Common components and the respective distributions describing their behaviour.

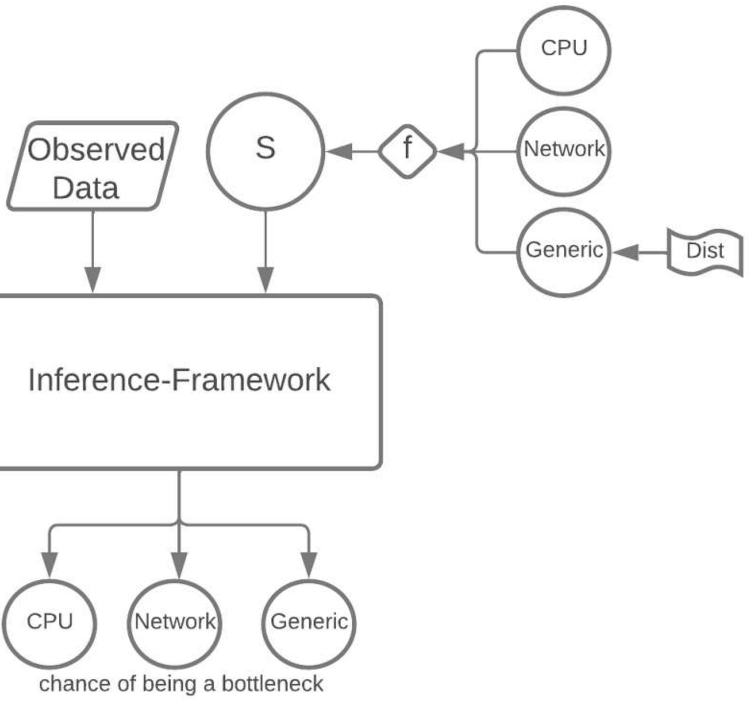


Figure 4.5: Overview of the inference framework with an example system as input.

take a tuple of mean response times as parameter and exhibit the predefined behaviour based on said means, the first one representing regular behaviour, while the second one represents behaviour under heavy-load. Table 4.1 shows these common components and the respective distributions describing their behaviour, along with the literature references used to define it. The framework takes the input and performs an MCMC inference to generate posteriors for each component. The resulting posterior distributions then reflect the probability of each component exhibiting bottleneck behaviour. We will look at specific examples and their results in Chapter 6.

4.5.1 Example

We continue with the video encoding example we used in previous chapters. First, lets specify our system more thoroughly by defining some pseudocode for the endpoint.

Algorithm 4.1: *Encode* endpoint

```
  Input: A URL to the raw video file
  Output: A URL to the encoded video
1 url ← read_url_from_request();
2 file ← load_video_asset(url);
3 data ← read_metadata(file);
4 configure_encoding();
5 output_url ← create_output();
6 while segment ← encode() do
7     write_to_output(segment);
8 end
9 return output_url;
```

Algorithm 4.1 shows the pseudocode for our *Encode* example system. It takes as an input a URL to a raw video file, which on Line 1 gets extracted from the request and is then downloaded on Line 2. Following that, lines 3 to 5 gather some data on the video and prepare configurations needed for the encoding and the output. Finally, lines 6 to 9 perform the actual encoding, writing the encoded video segments to the output as they get generated and finishing the whole process by returning a URL to the encoded video.

Now, since we know the algorithm of our endpoint of interest, we can start formulating the components. We will do so by looking closer at each line and categorizing them by the level at which their performance influences the overall performance of the endpoint. We start at the top-level, with the performance of Line 1 adding directly to the overall performance of the endpoint. Line 2 on the other hand already invokes the *load* component and thus depends on the second-level performance of the component. The next lines, 3, 4 and 5 are again top-level and add to the overall performance of the endpoint. Continuing with lines 6 and 7, we invoke the remaining components *encode* and *write* accordingly, each contributing to the response time via second-level performance. The algorithm invokes both components in a loop, but as neither one of the components' possible slower response times influences the response time of the other component within itself, there is no performance dependency between the components themselves. Finally, Line 9 concludes the process by returning the URL leading to the encoded video. We can now specify the system's components and performance dependencies using our categorization, by simply specifying one node for the top-level performance and one node for each component on the second-level. Since we only have dependencies from second-level to top-level, we simply add edges from each second-level node to the top-level node. Figure 4.6 showcases this basic performance dependency graph.

Finally, we can use the templates provided in the framework to actually model the system

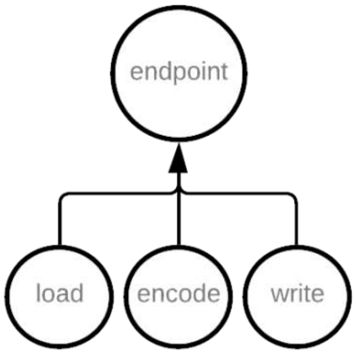


Figure 4.6: Performance dependency graph of the encoding example.

and its behaviour by simply translating the nodes of the performance dependency graph to components within the framework. We will do so in pseudocode. For a model specified using the actual implementation of the framework refer to Chapter 5.

Algorithm 4.2: Modelling *Encode*

```
1 load ← Network("load", (520, 780));
2 encode ← CPU("encode", (1200, 2400));
3 write ← ReadWrite("write", (700, 1000));
4 endpoint ← Generic("endpoint",
    component_times → N(sum(component_times), 100),
    [load, encode, write])
```

Algorithm 4.2 shows the definition of our model within the framework. Lines 1 to 3 specify each component and their behaviour using the templates *Network*, *CPU* and *ReadWrite* and providing a name and the tuple of mean response times respectively. Line 4 then instantiates the overall system "endpoint" and defines its behaviour as a function of its component response times, which samples from a Normal distribution at the sum of the components' response times and the previously estimated uncertainty. Lastly, we specify the list of components that "endpoint" depends on.

4.6 Performance distributions

To acquire realistic data for testing the implementation and to gain insight on the performance behaviour of different common component besides literature, we set up a test system that would allow us to generate data under semi-controlled conditions. *Semi*-controlled, since we wanted the data to resemble real data as closely as possible and we purposely let various factors like heat, computational power, caching and any computational overhead play a role during generation of the data. The system under test here was chosen to be a simple web-service that would expose different endpoints, performing either a high CPU-load task, a high network-load task, a high I/O (i.e. file read/write) task or a combination of those. This service was then run on a *Raspberry Pi Model 4* in a local network and put to test from another machine using the *Apache Benchmark Tool* [22].

Using this setup, the system was put to test for one day, sending a request to the endpoints every few seconds and measuring their response times. In parallel, we put load on the system coming from other sources than the requests or the web-service itself. In the case of CPU load, we gradually forced the CPU cores to maximum load by starting *yes* processes and redirecting them to */dev/null*, which causes the process to use up all available resources from a single CPU core. Network load was generated on the one side by downloading large files via the internet and on the other by uploading files to the test system from the other machine executing the tests via the local network. Generating I/O load then was done by copying large files from and to external storage devices via USB. We also generated load on all those components by performing real life tasks on the system in the form of updating and installing software. This was performed to get a sense of "real" data, the difference in performance of each of those components' performance under regular and heavy load and to gather testing data for the inference.

Figures 4.7, 4.8 and 4.9 show the response times of each tested endpoint as histograms respectively. Among a few outliers, one can clearly see the difference in response time between regular use and heavy-load (i.e. slowed down) use of the system. In case of the CPU component the distribution of response times under regular load as well as under heavy-load resembles a *Normal* distribution although slightly long-tailed. The distribution of the response times under normal load of the I/O and Network endpoints both seem to fit a *LogNormal* distribution, with their peak at the lower values and gradually declining density towards the longer tail in the higher response times. Also for both these endpoints, the distributions of response times under heavy load seem to follow the same type of distribution, namely a *Normal* distribution. These results seem fitting when considering that in regular use, firstly, it is hard for systems and components to perform below a certain minimal threshold and secondly, there can obviously be worse performance outliers forming a tail to the right, both nicely represented by a *LogNormal* distribution. On the other hand, under heavy-load it is reasonable that some executions/runs perform better and not just worse than the average performance, since the load might lessen and performance slightly shift towards regular load performance again, resulting in a *Normal* distribution.

Chen et. al [16] took a look at the performance distribution of computer systems, especially taking into account the traditional use of *Normal* distributions to model performance and testing its validity via benchmarks. They came to the conclusion that *Normal* distributions might not be fitting in general and they propose the use of *LogNormal* distributions to represent computer system's performance. Coming from a different approach, Khoshkbarforoushha and Ranjan [17] looked into predicting CPU performance distribution for *Hive queries* and their results coincide with those suggested by [16], predicting mainly *Normal*-like distributions with a longer tail. Based on these

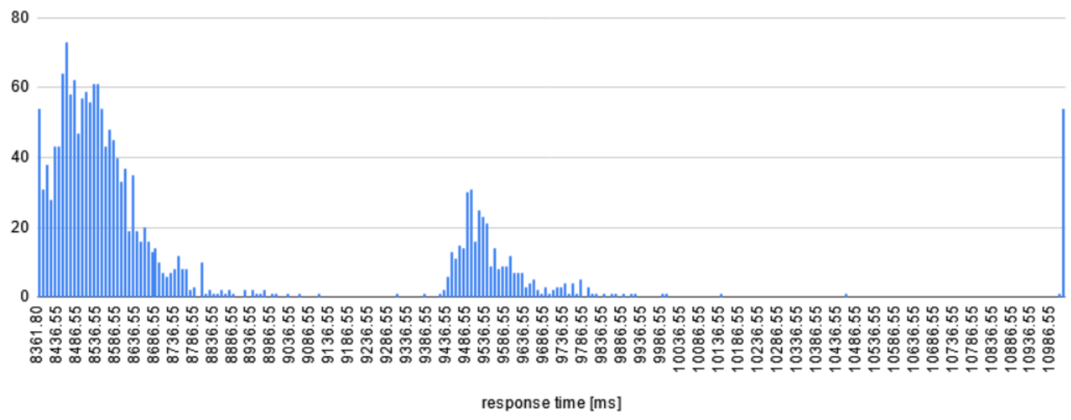


Figure 4.7: Histogram of response times of calls to the CPU endpoint.

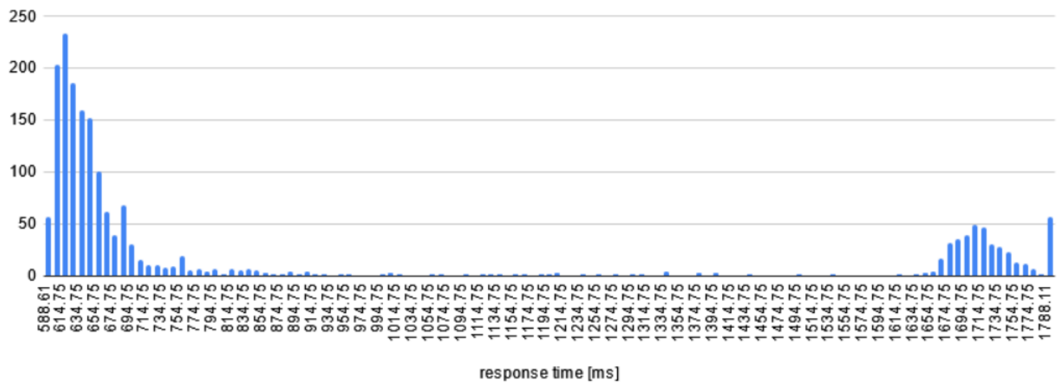


Figure 4.8: Histogram of response times of calls to the I/O endpoint.

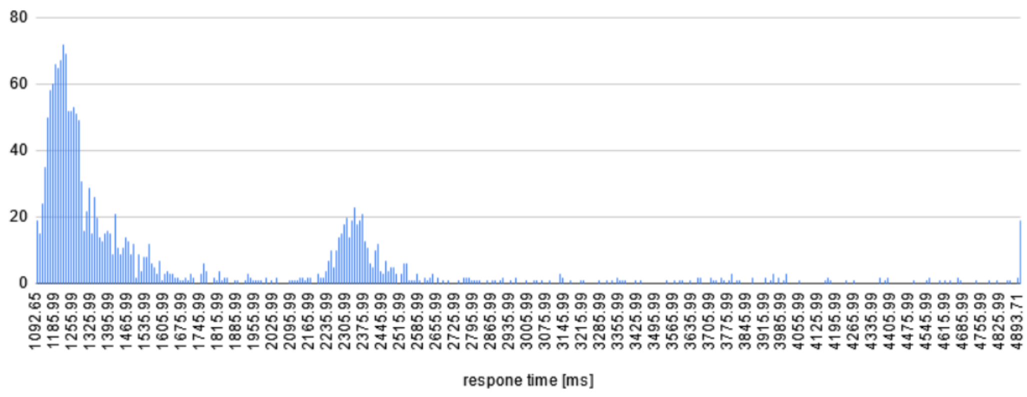


Figure 4.9: Histogram of response times of calls to the network endpoint.

papers, as well as our own observations, we presume a *LogNormal* distribution represents CPU performance behaviour. Looking at the performance of network components, Paxson [18] tested various analytical models to describe network connections and concluded that while modelling wide-area traffic exactly in a statistical way may not be possible, analytical models provide a good approximation. Further, they summarized their findings of analytical models best suited to describe various types of traffic, with *LogNormal* and *Pareto* distributions seemingly suiting best.

CHAPTER 5

Implementation

The implementation was done in *Python 3.8.10* using the package *Numpyro (version 0.11.0)*. The approach to the implementation was done in the same manner as described in Section 4, with the inference being automated by the *Numpyro* package. In this section, we will take a look at the actual model that was written and the inference that was performed on it. For the code discussed in the following subsections, these module imports are needed:

```python
import numpyro
import numpyro.distributions as dist
```

Further, for reproducibility and consistency in our results, we will use a fixed key as the seed for the random number generator and split further keys off of that as needed.

5.1 Model

In this section, we will look at the actual implemented model and the respective code. In general, the model reflects the model described in Section 4.3, differing just in details regarding the usage of the framework in *python*. So first, to recapitulate, we are again considering the system S, consisting of three components:

$$S = \{load, encode, write\}. \tag{5.1}$$

Now, as previously done in the example of the formal framework definition in Section 4.5.1, we specify our system via nodes and with the help of the framework's templates. We will define the model in the same manner as the pseudocode in Algorithm 4.2, again with slight adaptation for the *python* and *numpyro* specific implementation. Source code 5.1 shows the implementation, using python's `lambda` function to provide the behaviour of the components. For each of the sub-components *load*, *encode* and *write*, we specify their behaviour simply by returning a tuple of mean response times from the `lambda` function. The first element of the tuple is the mean response time of the component's regular behaviour, while the second element specifies the (estimated) mean response time under heavy-load. Lastly, the behaviour of the endpoint itself is captured via the `Generic` node, where we specify the list of sub-components and define the behaviour by using the sub-components' response times to define a Normal distribution in the `lambda`, which will be sampled from during inference. Further we set the flag `exclude` to `True` on this component since we do not want to consider it as a possible cause for the drop. The component templates can, in general, be further refined by specific arguments depending on which template is used. We will not go in detail on these arguments, as they can be seen in the implementation of the framework [15] itself.

```python
from components import CPU, Generic, Network, Queue, ReadWrite

def encoding_endpoint():
    load = Network('load', lambda _: (520, 780))
    encode = CPU('encode', lambda _: (1200, 2400))
    write = ReadWrite('write', lambda _: (700, 1000))
    return Generic('encoding_endpoint',
        lambda comp_times: dist.Normal(sum(comp_times), 100),
        components=[load, encode, write],
        exclude=True)
```

Source code 5.1: Model definition of *Encode*

We continue by preparing the model and the input data for the framework as seen in Source code 5.2. We simply get the `root_node` of the system we set up in Source code 5.1 and specify an input file containing the observed data. We make use of the *pandas* [23] package to read the `.csv` file and finally convert the retrieved data to a *numpy* array on Line 11.

```python
1  import numpy as np
2  import pandas as pd
3  from mcmc import InversePerformanceInference
4
5  def main():
6      root_node = encoding_endpoint()
7      inputfile = "./encoding_endpoint_data.csv"
8
9      print("Reading file", inputfile)
10     data = pd.read_csv(f'{inputfile}', delimiter=',', dtype=np.float32, header
       =None)
11     observed_times = np.array(data.values[0])
```

Source code 5.2: Model setup and input preparation

5.2 Inference

In this brief section, we will run the actual inference on our model above, given the observed response times described in Section 4.4 and using the framework as introduced in Section 4.5. For this, we will extend Source code 5.2 as follows. As shown in Source

```
1  import numpy as np
2  import pandas as pd
3  from mcmc import InversePerformanceInference
4
5  def main():
6      root_node = encoding_endpoint()
7      inputfile = "./encoding_endpoint_data.csv"
8
9      print("Reading file", inputfile)
10     data = pd.read_csv(f'{inputfile}', delimiter=',', dtype=np.float32, header
       =None)
11     observed_times = np.array(data.values[0])
12
13     num_steps = 300
14     num_chains = 3
15     num_warmup = 100
16
17     ipi = InversePerformanceInference(root_node, observed_times)
18     ipi.train(num_steps, num_warmup, num_chains)
19     ipi.get_results(plot=True, title=inputfile)
```

Source code 5.3: Framework setup and inference invocation

code 5.3, we import the framework and add lines 13 to 15 to define arguments for the MCMC inference, where `num_samples` refers to the number of samples generated within one Markov Chain and `num_chains` specifies the number of Markov Chains to be generated. Lastly, `num_warmup` refers to the number of warmup steps to be taken by the sampler, which is the amount of samples generated and then discarded before starting the actual inference. Next, we instantiate the framework itself, passing the `root_node` representing our model as first argument and the observed data `observed_times` as second argument on Line 17. With everything set up, we can then simply call the `train` method of the framework on Line 18 and pass in the arguments we defined for the inference. Once we have conditioned the model, we retrieve the results via the `get_results` method, where we can toggle plotting, specify a title for the plots and pass in a list of `sites` (i.e. components) we want to retrieve the results for. For the latter argument, we do not specify such list, since the default behaviour is to retrieve results for all components, which is want to do here anyways.

5.3 Results

Similar to the dataset previewed in Section 4.4, we have generated data for the system's behaviour while each of the remaining components display bottleneck behaviour to further test our model. We have run the inference with each dataset, looking to infer the probability of each component being a bottleneck. The mean and standard deviation of each component's chance of being the bottleneck (e.g. `load_slow_chance`) is displayed in Table 5.1 and their respective median is depicted in Table 5.2.

	`slow_chance` mean			`slow_chance` stddev		
dataset	*load*	*encode*	*write*	*load*	*encode*	*write*
load	0.4380	0.3418	0.5595	0.2805	0.2384	0.2865
encode	0.5498	0.6735	0.3418	0.2859	0.2457	0.2357
write	0.5528	0.3327	0.4484	0.2906	0.2421	0.2827

Table 5.1: Mean and standard deviation of each component's `slow_chance`

	`slow_chance` median		
dataset	*load*	*encode*	*write*
load	0.4084	0.3001	0.5924
encode	0.5678	0.7327	0.3015
write	0.5726	0.2823	0.4232

Table 5.2: Median of each component's `slow_chance`

To further visualise the results and observe their development across the inference, we can plot a histogram for each component's `slow_chance` and the trace of it during the inference. The figures Figure 5.1, Figure 5.2 and Figure 5.3 display these plots respectively for component *load*, *encode* and *write* each demonstrating bottleneck behaviour.

Looking at the histograms we can see the results reflected pretty clearly. The histograms of the results of *load* itself being a bottleneck, in Figure 5.1, show that the inference tends towards *write* being the cause of the performance drop, but with a notable amount of uncertainty represented by the gradual increase instead of a steep incline towards the right of the plot. We can also see some uncertainty in the low probability of *load* having caused the drop, compared to the certainty in the *encode*-histogram with a steep decline. In the next group of histograms in Figure 5.2, for *encode* acting as the bottleneck, we can clearly see the inference predicting *encode* as the possible cause, but we can also see the possibility of *load* having caused the performance drop, although with the uncertainty dampening that result as well. The third histogram triple, seen in Figure 5.3, displays the results obtained while *write* exhibits bottleneck behaviour and does in fact reflect the inference's wrong result of *load* being the cause, but again with quite some uncertainty. Also, similar to the histograms for the first dataset, we can see uncertainty in the low probability of *write* being cause.

So the inference predicts *load* and *write* as exhibiting heavy-load behaviour on the wrong dataset each. Why would that be the case? As mentioned earlier in Chapter 4, the response times for *load* and *write*, and thus also its datasets, lie very close to each other, so it makes sense that the difference between those two components gets blurry,

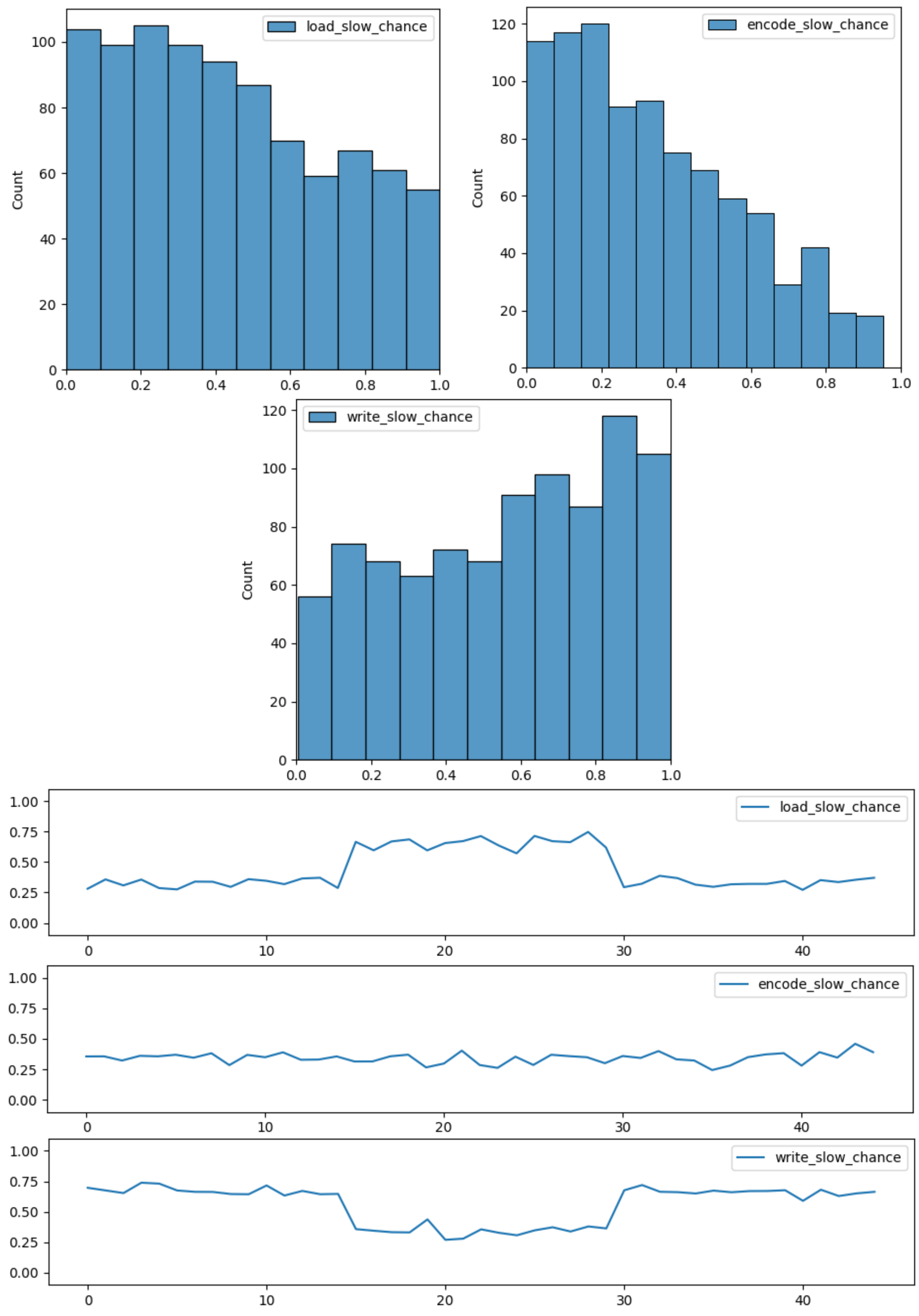


Figure 5.1: Histograms and traces of each component's `slow_chance` for component *load* showing bottleneck behaviour.

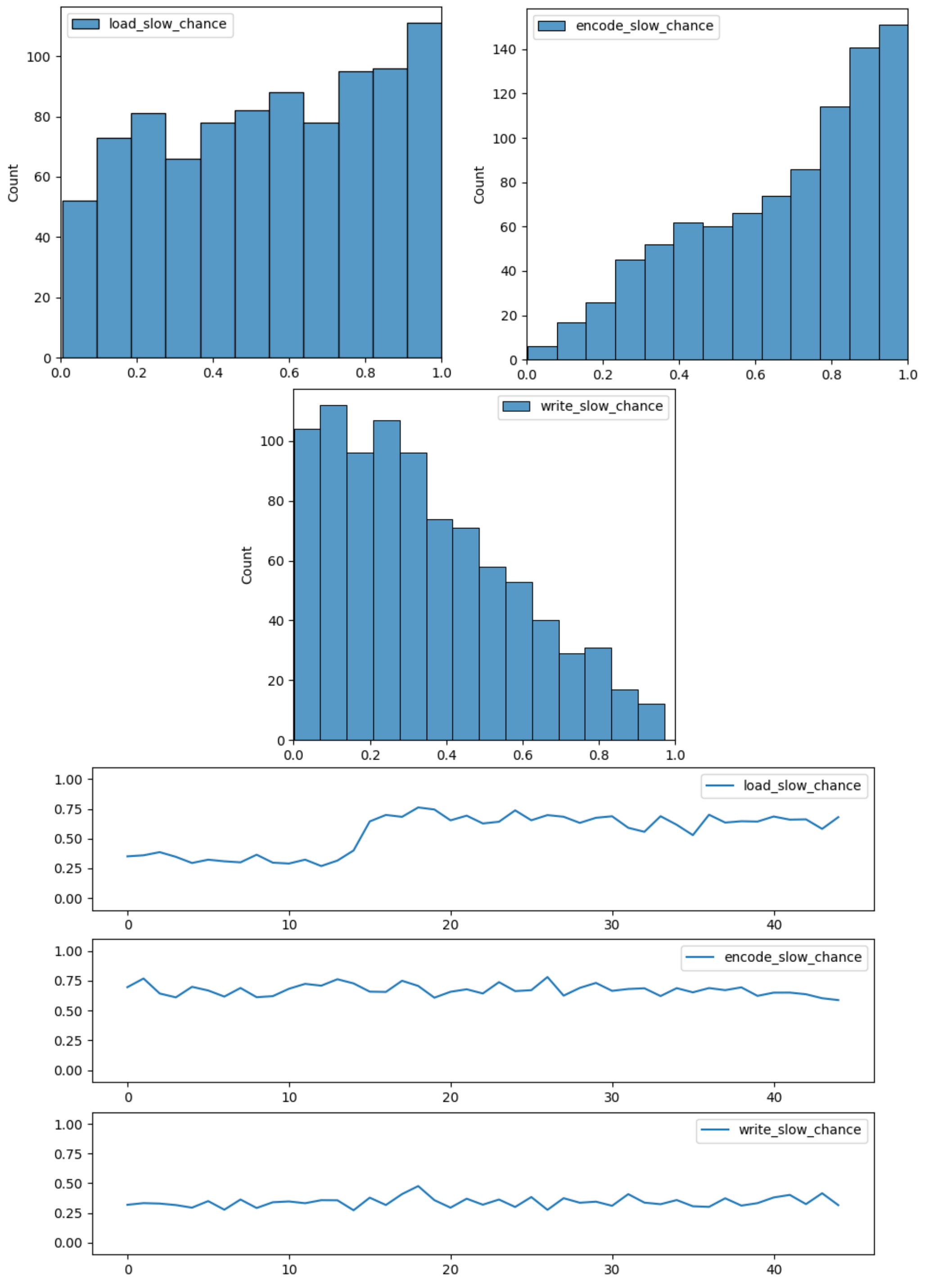


Figure 5.2: Histograms and traces of each component's `slow_chance` for component *encode* showing bottleneck behaviour.

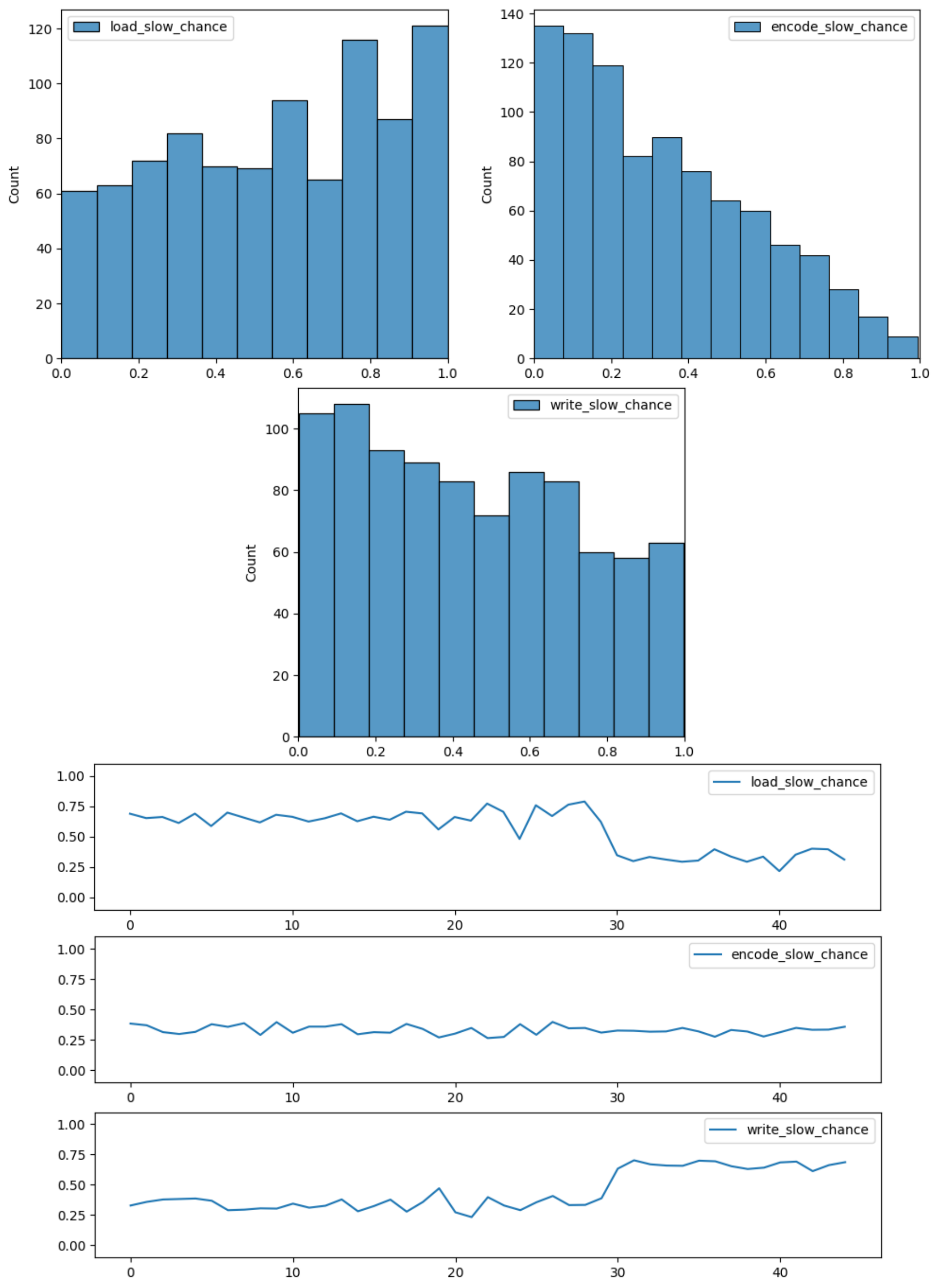


Figure 5.3: Histograms and traces of each component's `slow_chance` for component *write* showing bottleneck behaviour.

especially considering the additional noise coming from sampling and from the endpoint's performance.

For further insight we can take a look at the traces, which depict the development of each component's `slow_chance` as the inference progresses along the Markov chains. In Figure 5.1 we can see the traces for the run in which *load* exhibited bottleneck behaviour. The trace of component *encode* is not too interesting, with the trace remaining steadily below 0.5, but the traces for *load* and *write* give interesting insight into why the results are uncertain. Looking at the trace of *load*, in the second chain the probability increases drastically and jumps above 0.5, although it then drops again in the third chain. Quite contrary, the trace for *write* starts high and then drops drastically below 0.5 in the second chain, just to increase again in the third. This might just be related to the number of chains generated and in fact, running the inference with five or even seven chains, results in the traces continuing to increase and decrease drastically, leaving the probabilities to converge around 0.5 for both components, indicating indifference in these cases. Now in the second set of traces, we can clearly see *encode* steadily above 0.5 and again *load* experiencing the jump in the second chain, this time maintaining it across the third, which also explains the positive skew of the histogram. Lastly in the third trace, showing the development of `write_slow_chance`, we can observe quite the opposite of the behaviour seen in the traces of the *load* dataset. The trace of the *load* component starts at a higher value and drops below 0.5 for the third chain, while the trace of the *write* component mainly remains in the lower half until it drastically increases to values above 0.5 in the third chain. And in fact, also increasing the number of chains for inference on this dataset results in the same convergence at around 0.5 we observed for the *load* dataset.

CHAPTER 6

Evaluation

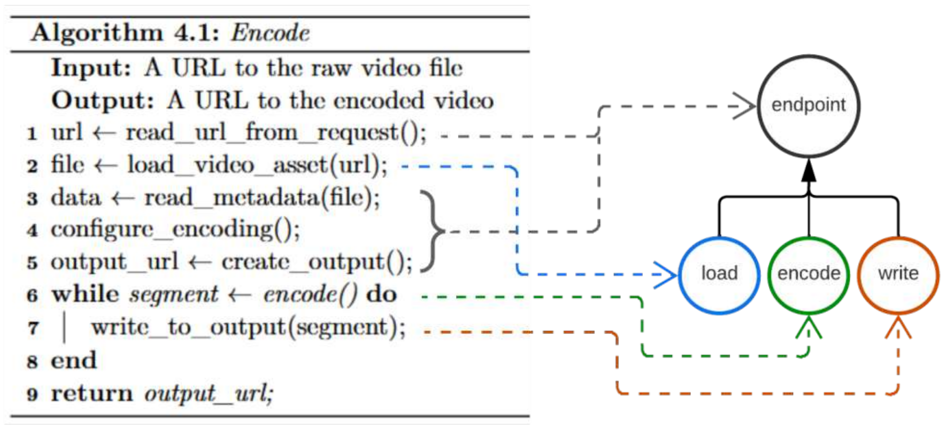
This chapter will, in Section 6.1, look into three artificial test systems and their different characteristics regarding performance distribution as well as structure and finally propose models for each of them, formulated using the framework. These models will then be implemented in Section 6.2, where we will also present the setup of the inference and the results in terms of run time as well as correctness for each system. Lastly, we will look into a case study of a real service, evaluating the modelling process and the inference using data gathered from benchmarking the service in the local network.

6.1 Systems

6.1.1 System 1: Encode

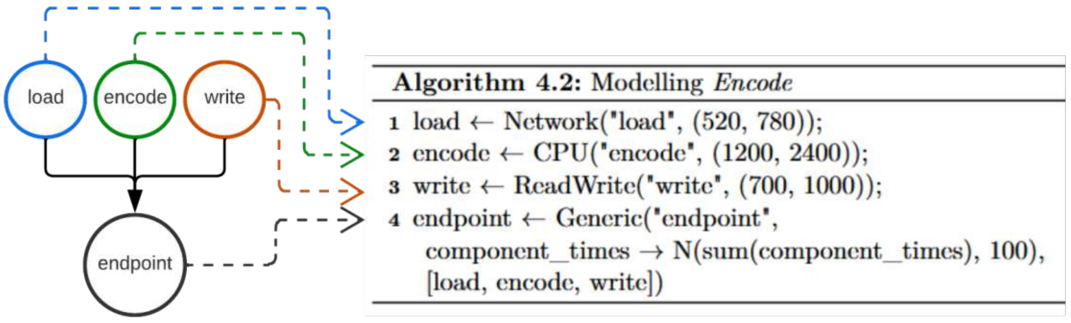
The first system is *Encode*, the system we constructed during the course of this thesis. Although, we already explored this system, we will use it to illustrate the whole modelling process and to enhance it to a model that represents real behaviour by assuming more realistic response time behaviour. As an overview, Figure 6.1 shows how the code of the endpoint is translated to a graph representing performance dependencies and Figure 6.2 shows how said graph is then mapped to a model within the framework. As mentioned previously, the system consists of three components, `load`, `encode` and `write`, which each exhibit different performance behaviour. Component `load` transfers a (possibly large) video file via wide area network, `encode` is a CPU-heavy process that encodes a raw video into segments, and `write` writes those segments to a predefined output storage, meaning it exhibits I/O performance behaviour.

So for modelling the system, we identify groups of lines in the code that contribute to either each of the components or to the overall performance of the endpoint. As we can see in Figure 6.1, we mainly have one line per component, invoking the respective process of loading, encoding or writing, while the remaining lines of code contribute solely



```
Algorithm 4.1: Encode
  Input: A URL to the raw video file
  Output: A URL to the encoded video
  url ← read_url_from_request();
  file ← load_video_asset(url);
  data ← read_metadata(file);
  configure_encoding();
  output_url ← create_output();
  while segment ← encode() do
  |   write_to_output(segment);
  end
  return output_url;
```

Figure 6.1: Translating the *Encode* endpoint's code to a performance dependency graph.



```
Algorithm 4.2: Modelling Encode
  load ← Network("load", (520, 780));
  encode ← CPU("encode", (1200, 2400));
  write ← ReadWrite("write", (700, 1000));
  endpoint ← Generic("endpoint",
    component_times → N(sum(component_times), 100),
    [load, encode, write])
```

Figure 6.2: Mapping the performance dependency graph of *Encode* to a probabilistic model.

to the overall performance and do not invoke any sub-behaviour. From these groups we can formulate the *nodes* for the performance dependency graph and insert edges representing the performance dependencies. Now the last step is to translate this graph into a model within the framework. For that we simply choose a template or create a custom `Generic` node for every node on the graph as seen in Figure 6.2. Finally we make sure the dependencies (i.e. edges between nodes) are correctly set by passing the list of sub-components to the respective parent component.

As mentioned before, we will update the behaviour of this model to be more representative of a real system. We do so mainly by updating the mean response times that are passed to the component templates. The resulting model now looks as shown in Source code 6.1. Now that we have defined our model, we still need some observed data that we can use for testing. We will generate multiple datasets containing the system's overall response times, one for each inner component exhibiting heavy-load behaviour. The generation of those response times is done by explicitly setting one of the components' `is_slow` flag to

```python
def system1_encode():
    asset_download = Network('download', lambda _: (720, 1100))
    encoding = CPU('encoding', lambda _: (1200, 2400))
    output_write = ReadWrite('write', lambda _: (300, 600))
    return Generic('encode_endpoint',
        lambda comp_times: dist.Normal(sum(comp_times), 33),
        components=[asset_download, encoding, output_write],
        exclude=True)
```

Source code 6.1: Implemented model of system *Encode*

`True` and then generating $n \in \mathbb{N}$ samples by sampling from the root node's distribution. Source code 6.2 shows the (simplified) process of generating the "observed" data.

```python
def set_slow_components(node: Generic, slow_components: List[str], visited:
    List[Generic] = []) -> None:
    if node not in visited and node.exclude == False:
        node.set_slow(node.name in slow_components)
        visited.append(node)
    for comp in node.components:
        set_slow_components(comp, slow_components, visited)

num_samples = 1000
times = []
set_slow_components(system1_encode(), ['write'])
for _ in range(0, num_samples):
    time = numpyro.sample('response_time', root.get_dist())
    times.append(time.item())
```

Source code 6.2: Generating response time data for the system *Encode*

6.1.2 System 2: Pop

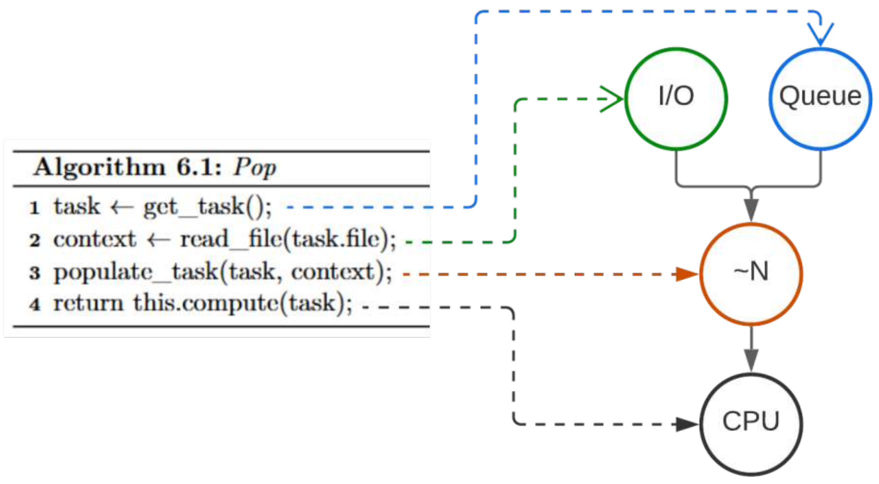
The next system is representative of a service that works with the dependency of a queue. The system itself receives a task from the queue, populates the task according to a related file which is read from the system storage and finally computes the task in a CPU heavy computation.

Let us first take a look at the pseudocode for system *Pop*. We can see the code of *Pop* in

Algorithm 6.1: *Pop*

```
1 task ← get_task();
2 context ← read_file(task.file);
3 populate_task(task, context);
4 return this.compute(task);
```

Algorithm 6.1. Line 1 already is a blocking call to a sub-component, namely a queue, from which the system gets its next task once it has arrived in the queue. The next line,



Algorithm 6.1: *Pop*

```
1 task ← get_task();
2 context ← read_file(task.file);
3 populate_task(task, context);
4 return this.compute(task);
```

Figure 6.3: Translating the *Pop* endpoint's code to a performance dependency graph.

Line 2, again invokes a sub-component, this time for reading a file related to the task and containing contextual data needed to compute the task. This file is stored within the system. Continuing, Line 3 populates the task with the relevant context read from the file, which can be considered a rather trivial computation, not invoking a sub-component. Lastly, Line 4 finally computes the task and resolves the request, although not invoking another sub-component but performing the CPU-heavy computation as the endpoint itself. From the general knowledge of the system and its code, we can now proceed by grouping and drawing the performance dependency graph. The resulting graph, along with the respective lines of code that relate to the nodes is shown in Figure 6.3. We can see how this system differs in its performance dependencies to the system describe in Section 6.1.1 due to a third layer that gets aggregated on the second, roughly by a Normal distribution, and additionally due the overall endpoint also behaving like a CPU component.

Now finally, let us model the system in *python* using the framework. We, again, use templates to translate the graph nodes into nodes of the framework. We can identify the performance groups' behaviour as `Queue` and `ReadWrite`, with both of these getting aggregated within a Normal distribution and lastly we can identify the endpoints behaviour as `CPU`. Putting this to code leaves us with the model displayed in Source code 6.3. We use the response times of `queue` and `readfile` in the `setup` component to sample from a *Normal* distribution around the sum of the sub-components' response times. Lastly we specify the `pop_endpoint` component with CPU behaviour and the base provided by the response time of `setup`.

Again, we generated "observed" data for this system and each of the components (including the root component) exhibiting heavy-load behaviour.

```python
def system2_pop():
    task_queue = Queue('queue', lambda _: (100, 200))
    file_read = ReadWrite('readfile', lambda _: (600, 1000))
    setup = Generic('setup',
                    lambda comp_samples: dist.Normal(comp_samples[0] +
    comp_samples[1], 100),
                    components=[task_queue, file_read],
                    exclude=True)
    return CPU('pop_endpoint', lambda comp_samples: (comp_samples[0],
    comp_samples[0]*2.0), components=[setup])
```

Source code 6.3: Implemented model of system *Pop*

6.1.3 System 3: Convert

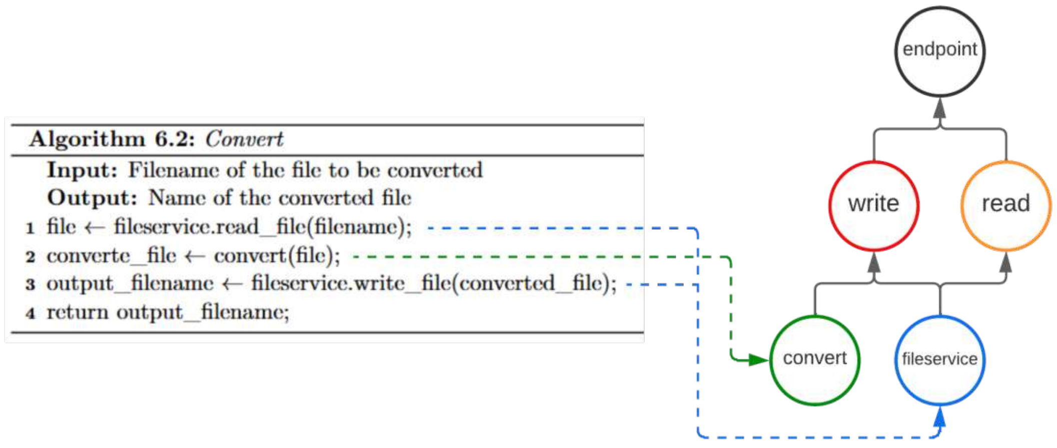
The last system that we will define, called *Convert*, will incorporate the use of a shared service twice within one execution of the endpoint and will also offer a more interesting dependency graph. The service first reads a file from the filesystem, converts said file into a different format and then writes the new file onto the filesystem again. This incorporates two invocations of a `fileservice` and one other component doing the conversion.

Again, let us start by looking at the pseudocode. Looking at the code of *Convert* in

Algorithm 6.2: *Convert*

```
Input: Filename of the file to be converted
Output: Name of the converted file
1 file ← fileservice.read_file(filename);
2 convert_file ← convert(file);
3 output_filename ← fileservice.write_file(converted_file);
4 return output_filename;
```

Algorithm 6.2, we can see more clearly its dual use of the `fileservice`, once on Line 1 for reading and once on Line 3 for writing. In between that, on Line 2, the endpoint invokes a CPU-heavy conversion component to perform the actual file conversion. Finally, Line 4 concludes the endpoint's execution by returning the filename of the converted file. Now since we use `fileservice` twice and we only want to infer its `slow_chance` once, we cannot define two separate nodes to represent this component, meaning we define it once and then make use of it twice (similar to instantiation and usage of a service in object-oriented programming). So for the performance dependency graph, we first create the `fileservice` node and then two separate nodes, one for `read` and `write` each, which depend on the `fileservice` node for their performance. Further, there is a second dependency for the `write` node, namely the conversion, so we introduce another `convert` node as a dependency for `write`. Lastly, we combine it all via the `endpoint` node, which depends on both, the `write` and `read` node. The resulting graph along with the mapped lines of code can be seen in Figure 6.4.



```
Algorithm 6.2: Convert
   Input: Filename of the file to be converted
   Output: Name of the converted file
1  file ← fileservice.read_file(filename);
2  converte_file ← convert(file);
3  output_filename ← fileservice.write_file(converted_file);
4  return output_filename;
```

Figure 6.4: Translating the *Convert* endpoint's code to a performance dependency graph.

As a final step, we again convert the graph to an actual model within the framework by using the templates and defining the nodes of the system. We use the `ReadWrite` template for the `fileservice` node and the `CPU` template for the `convert` node. For `write`, `read` and `endpoint` we make use of the `Generic` node to define custom behaviour and since those components simply aggregate the performances of their sub-components, we will describe their behaviour with *Normal* distributions and exclude them from `slow_chance` inference. The resulting model implementation can be seen in Source code 6.4. In the same manner as for the systems above, we generated "observed"

```python
def system3_convert():
    fileservice = ReadWrite('fileservice', lambda _: (500, 700))
    convert = CPU('convert', lambda _: (200, 400))

    read_file = Generic('read',
        lambda comp_times: dist.Normal(comp_times[0], 10),
        components=[fileservice],
        exclude=True)
    write_file = Generic('write',
        lambda comp_times: dist.Normal(sum(comp_times) 10),
        components=[convert, fileservice],
        exclude=True)

    return Generic('endpoint',
        lambda comp_times: dist.Normal(sum(comp_times), 25),
        components=[read_file, write_file],
        exclude=True)
```

Source code 6.4: Implemented model of system *Convert*

data that will be used for testing the inference.

6.2 Inference & Results

Now that we have defined our test systems and models in Section 6.1, we can start with the inference and evaluate its quality and run time. This section will cover each system separately by presenting the inference parameters used and then evaluating the inference and its results. Inference is implemented as shown in Source code 6.5. Since we are also measuring the run time of the inference, we will not plot the graphs immediately after completion of the inference but rather manually trigger the plotting once the run time was measured to avoid blocking the process until user interaction. For each system, we will adapt the values `num_samples`, `num_warmup` and `num_chains` to find the best results. The inference was performed on a laptop with an *AMD Ryzen 9 5900X* (3.3 GHz) CPU and 16 GB of RAM.

```python
start = datetime.now()

ipi = InversePerformanceInference(root_node, observed_times)
ipi.train(num_samples, num_warmup, num_chains)
samples, _, sites = ipi.get_results(plot=False)

end = datetime.now()
print("Runtime:", (end-start))

ipi.plot_samples_for_sites(samples, sites, f"{inputfile} Histograms")
ipi.plot_traces_for_sites(samples, sites, f"{inputfile} Traces")
```

Source code 6.5: Inference of each system

6.2.1 System 1: Encode

Starting inference on this system, we will use a similar structure to define the parameters for the inference algorithm as for the example in Section 4.5.1, but since we also have a larger number of observed response times ($n = 1000$) we will also increase our number of samples and warmup samples generated during inference. We define the parameters for this inference to be:

$$\begin{aligned}
\texttt{root\_node} &= \texttt{system1\_encode()}\\
\texttt{num\_samples} &= \texttt{3000}\\
\texttt{num\_chains} &= \texttt{3}\\
\texttt{num\_warmup} &= \texttt{1000}
\end{aligned} \tag{6.1}$$

Performing the inference for each of the datasets generated in Section 6.1, produced the mean and standard deviations shown in Table 6.1 with the respective median shown in Table 6.2. The run time of the inference for each dataset is shown in Table 6.3

In an optimal result, we would see the highest mean and median values at the diagonal of the table. As we can see from the results, dataset `download` breaks this proposal, as

	slow_chance mean			slow_chance stddev		
dataset	*download*	*encoding*	*write*	*download*	*encoding*	*write*
download	0.4408	0.3345	0.5530	0.2835	0.2417	0.2873
encoding	0.5549	0.6658	0.4429	0.2821	0.2361	0.2825
write	0.4426	0.3318	0.5527	0.2801	0.2349	0.2840

Table 6.1: Mean and standard deviation of each component's `slow_chance` of system *Encode*.

	slow_chance median		
dataset	*download*	*encoding*	*write*
download	0.4091	0.2874	0.5786
encoding	0.5793	0.7016	0.4144
write	0.4197	0.2893	0.5733

Table 6.2: Median of each component's `slow_chance` of system *Encode*

	dataset		
	download	*encoding*	*write*
runtime [hh:mm:ss:SSS]	0:03:36.206	0:03:51.682	0:04:00.840

Table 6.3: Run time of the inference on each dataset for system *Encode*.

for that dataset the inference suggests a higher probability for the `write` component to have caused the observed data, instead of the `download` component. We tested increasing the number of chains to 5, 7 and 10 to observe any change in convergence, which produced mixed results. The prediction of `encoding` for the `encoding` dataset stayed at roughly the same percentage across all number of chains while the probabilities of `download` and `write` on their respective datasets seemed to converge towards the same value at roughly 0.47 with 7 chains. From there on they started diverging again in runs with 10 chains, moving towards the same values that were produced when running 3 chains. With increasing number of chains the run times also significantly increased, the specific times for each inference can be seen in Table 6.4.

	run time of dataset [hh:mm:ss:SSS]		
`num_chains`	*download*	*encoding*	*write*
5 chains	0:07:09.889	0:08:21.400	0:08:49.685
7 chains	0:10:11.032	0:10:51.323	0:11:03.942
10 chains	0:14:02.140	0:17:32.782	0:13:26.351

Table 6.4: Run time of the inference performed with 5, 7 and 10 Markov chains on each dataset for system *Encode*.

On the other hand, what does seem to improve the results for this system is increasing the *target acceptance probability* of the MCMC inference from 0.8 to 0.9, which can be done via the framework's `train` call as seen in Source code 6.6. Although the results for the encoding and write datasets remain the same, it no longer clearly predicts the `write` component on the `download` dataset, unfortunately it also does not correctly infer the `download` component but rather states the roughly same probability of 0.44 for each of the two components to have caused the data seen in the `download` dataset.

We can see some clear indifference between the `download` and `write` components, while on the other hand a performance drop caused by the `encoding` component seems to be reliably inferred. One explanation for this might be the large difference in the mean response time of the CPU-heavy `encoding` component compared to the response time means of the network and I/O heavy components, making a performance shift of the CPU-heavy component more easily detectable among the data. Another reason could stem from the fact that when looking at the mean and standard deviation of the observed data, one notices quite a high standard deviation, around four to six hundred, on the download and write datasets. This is more than the difference of the actual mean response times of these components and could cause indifference when trying to infer if either of these components' exhibited heavy-load behaviour. We tested the latter by decreasing the estimated behaviour of the `write` component to $(200, 400)$, for regular and heavy-load behaviour respectively, and ran the inference with adapted "observed" data, which resulted in seeing the same behaviour as observed in Section 5.3, with the results of `download` and `write` converging towards indifference.

```
ipi.train(num_samples, num_warmup, num_chains, target_accept_prob=0.9)
```

Source code 6.6: Setting `target_accept_prob` for the inference.

6.2.2 System 2: Pop

For comparability and considering the size of the datasets for this system is again $n = 1000$, we will start with the same inference parameters as in Section 6.2.1. So we will use the following parameters:

```
root_node = system2_pop()
num_samples = 3000
num_chains = 3
num_warmup = 1000
```
(6.2)

Again, in the same manner as in Section 6.2.1, we executed the inference and measured its run time. The mean and standard deviation can be seen in Table 6.5 and the median in Table 6.6. Run times are presented in Table 6.7.

We can see the inference mainly predicting the `pop_endpoint` component as the cause for the observed data across all datasets, while being more uncertain on the `queue` and

	slow_chance mean			slow_chance stddev		
dataset	*pop_endpoint*	*queue*	*readfile*	*pop_endpoint*	*queue*	*readfile*
pop_endpoint	0.6659	0.4423	0.4449	0.2337	0.2828	0.2818
queue	0.5585	0.3306	0.3317	0.2816	0.2321	0.2367
readfile	0.5554	0.4434	0.4429	0.2815	0.2807	0.2828

Table 6.5: Mean and standard deviation of each component's slow_chance of system *Pop*.

	slow_chance median		
dataset	*pop_endpoint*	*queue*	*readfile*
pop_endpoint	0.7044	0.4125	0.4198
queue	0.5896	0.2894	0.2901
readfile	0.5862	0.4127	0.4209

Table 6.6: Median of each component's slow_chance of system *Pop*.

	run time of dataset [hh:mm:ss:SSS]		
num_chains	*download*	*encoding*	*write*
3 chains	0:03:41.640	0:03:47.575	0:03:52.348
5 chains	0:08:31.580	0:07:19.964	0:07:55.242
7 chains	0:10:03.141	0:06:51.236	0:08:44.625

Table 6.7: Run times of the inference on each dataset for system *Pop*

readfile dataset. The histograms in Figure 6.5 reflect these results respectively, with the broader curves showing uncertainty. Now the traces for this run, as seen in Figure 6.6 do not provide any further insight and rather simply reflect the results, so we increased the number of chains and performed the inference again. We ran the inference again with 5, 7 and 10 chains, getting the best results with 5 chains. The run times for these runs can also be seen in Table 6.7. When looking at the histograms of the run with 5 chains, shown in Figure 6.7, one can clearly see more confindence in all results, but with more involvement of queue on each dataset. The exact probabilities are shown in Table 6.8 with the means displayed in Table 6.9.

	slow_chance mean			slow_chance stddev		
dataset	*pop_endpoint*	*queue*	*readfile*	*pop_endpoint*	*queue*	*readfile*
pop_endpoint	0.6695	0.5984	0.3340	0.2368	0.2715	0.2359
queue	0.4681	0.5343	0.3306	0.2892	0.2864	0.2343
readfile	0.4680	0.5999	0.5330	0.2876	0.2700	0.2848

Table 6.8: Mean and standard deviation of each component's slow_chance of system *Pop* for inference with 5 chains.

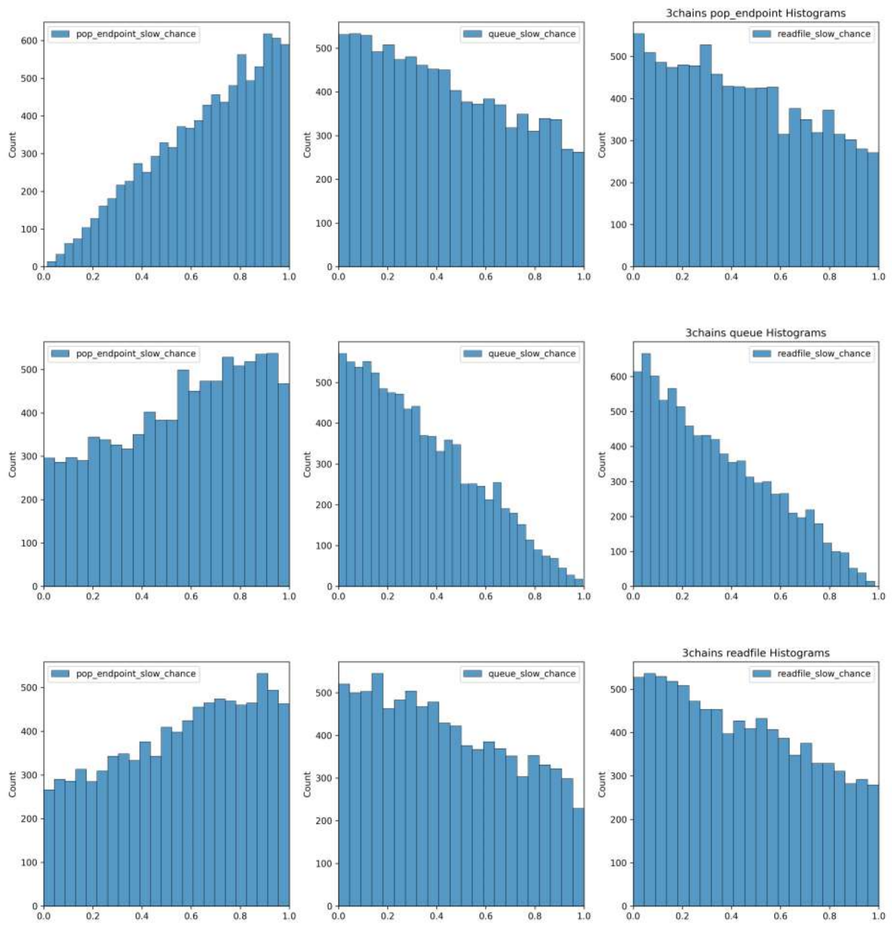


Figure 6.5: Histograms for inference on system *Pop* using 3 chains.

As we can see, the prediction for dataset `pop_endpoint` is rather confident and correct, most likely due to the high impact on performance of that component, similar to the result of Section 6.2.1. For the second dataset, with component `queue` having shown heavy-load behaviour, the inference does result in the right component having the highest probability, but with a significant uncertainty as seen in Figure 6.7, resulting in a probability of just 0.55. For the last dataset, in which `readfile` had exhibited heavy-load behaviour, the results suggest `queue` as the culprit again, this time even with more confidence at 0.63, but also open the possibility of `readfile` being the cause, although with uncertainty and at just 0.54. Interestingly, when running the inference with 10 chains on the `readfile` dataset, the results for each component seem to converge at around 0.5, suggesting that a performance drop caused by component `readfile` might hardly be inferrable in this setting. We again, tested some slightly different variations of defined performance behaviour for this system on respectively generated datasets, which all amounted to roughly the same results, varying just in the exact probabilities but not

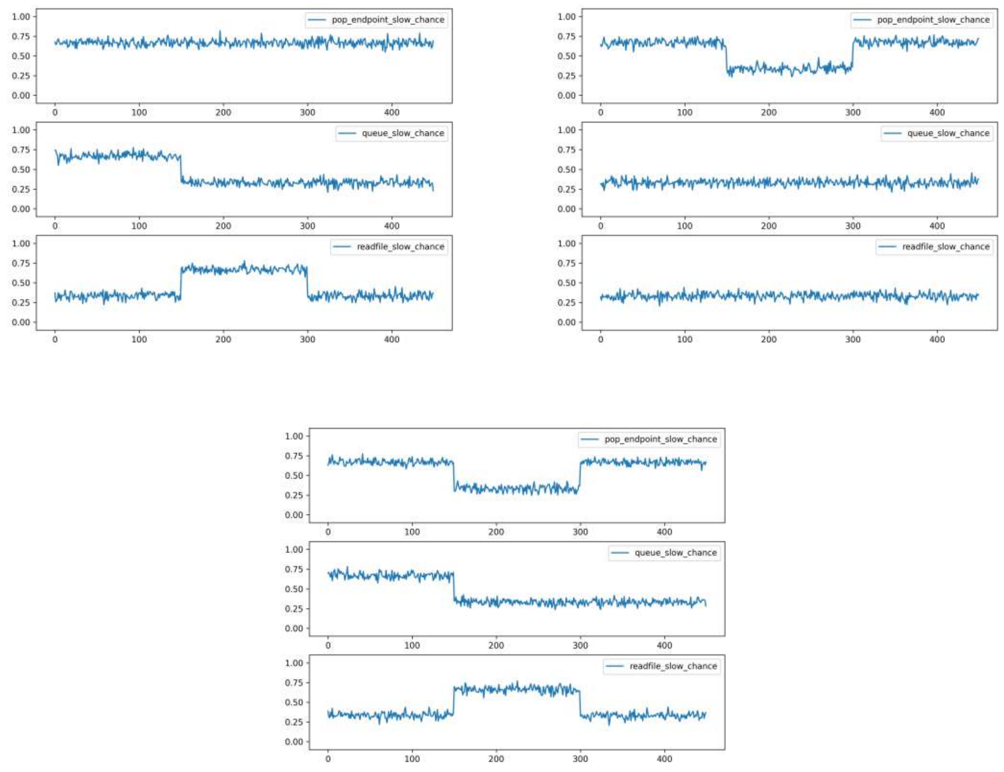


Figure 6.6: Traces of the inference on system *Pop* using 3 chains.

	`slow_chance` median		
dataset	*pop_endpoint*	*queue*	*readfile*
pop_endpoint	0.7135	0.6355	0.2938
queue	0.4492	0.5508	0.2910
readfile	0.4520	0.6358	0.5465

Table 6.9: Median of each component's `slow_chance` of system *Pop* for inference with 5 chains.

much in the gained information.

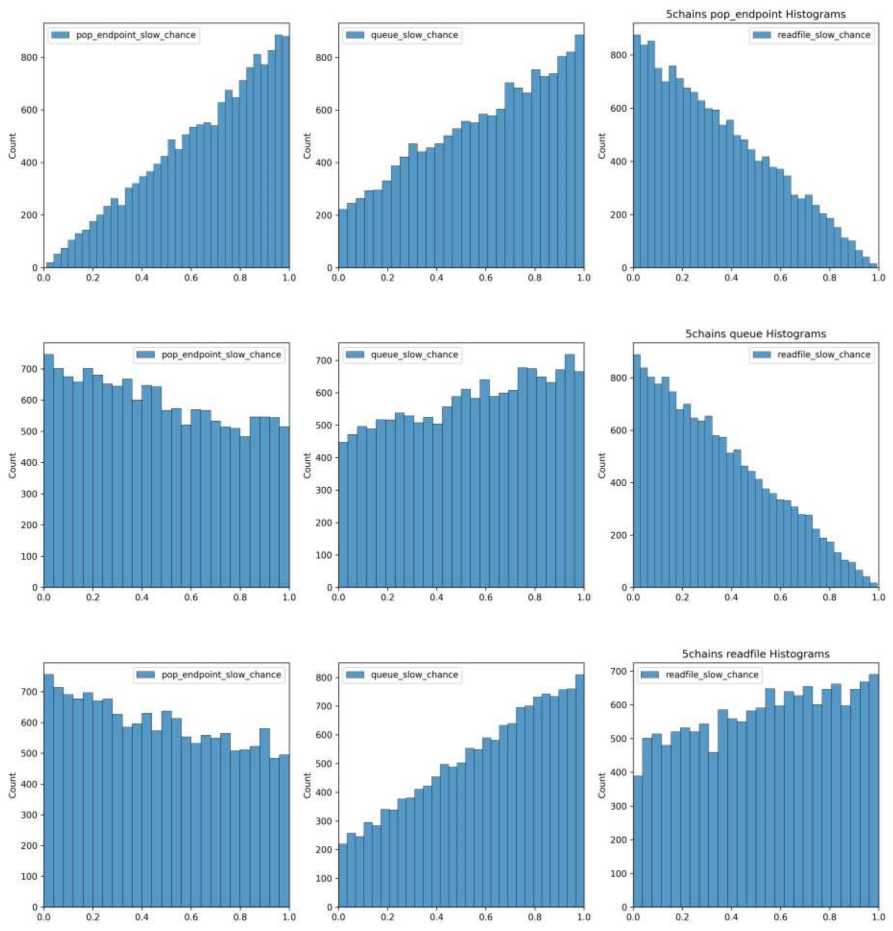


Figure 6.7: Histograms for inference on system *Pop* using 5 chains.

6.2.3 System 3: Convert

In the same manner as before, we start inference with the following parameters:

```
root_node = system3_convert()
num_samples = 3000
num_chains = 3                                  (6.3)
num_warmup = 1000
```

This yields the mean, standard deviation and medians seen in tables 6.10 and 6.11, with the run times displayed in Table 6.12. These initial results seem to not provide any useful information since the result of the inference on each dataset is essentially the same, but as we will see, these results are a nice showcase of the importance of tweaking the inference's parameters.

	`slow_chance` mean		`slow_chance` stddev	
dataset	*fileservice*	*convert*	*fileservice*	*convert*
fileservice	0.5510	0.4506	0.2821	0.2875
convert	0.5541	0.4502	0.2825	0.2844

Table 6.10: Mean and standard deviation of each component's `slow_chance` of system *Convert*.

	`slow_chance` median	
dataset	*fileservice*	*convert*
fileservice	0.5747	0.4258
convert	0.5793	0.4241

Table 6.11: Median of each component's `slow_chance` of system *Convert*.

	run time of dataset [hh:mm:ss:SSS]	
`num_chains`	*fileservice*	*convert*
3 chains	0:14:08.869	0:13:48.346
5 chains	0:18:45.354	0:17:43.022
7 chains	0:35:59.408	0:34:03.561

Table 6.12: Run times of the inference on each dataset for system *Convert*

We again, tested the inference with 5 and 7 chains, unfortunately without getting any clear improvement on the results, instead only increasing run time. Now to reduce run time, we started decreasing the number of samples and warmup samples generated. We started seeing an improvement at 1000 samples which settled at 300. Further improvement was then gained by reducing the number of warmup samples, getting the best results at `num_warmup=100`. Even further decrease resulted in less reliable results. Now the drastic decrease in sampling allowed us to increase the number of chains again, although we stopped seeing improvement in the results already at 12 chains, getting the best results with 10 chains. These results can be seen in Table 6.13, with the respective medians in Table 6.14 and the run times for 10 chains in Table 6.15.

This test system acts as a nice example showcasing the importance of parameter tweaking for the inference, also showcasing that a mindful reduction of generated samples can still

	`slow_chance` mean		`slow_chance` stddev	
dataset	*fileservice*	*convert*	*fileservice*	*convert*
fileservice	0.5859	0.5380	0.2773	0.2830
convert	0.4653	0.5351	0.2868	0.2877

Table 6.13: Mean and standard deviation of each component's `slow_chance` of system *Convert* with 10 chains, 300 samples and 100 warmup samples.

	`slow_chance` median	
dataset	*fileservice*	*convert*
fileservice	0.6196	0.5587
convert	0.4503	0.5529

Table 6.14: Median of each component's `slow_chance` of system *Convert* with 10 chains, 300 samples and 100 warmup samples.

	run time of dataset [hh:mm:ss:SSS]	
`num_chains`, `num_samples`	*fileservice*	*convert*
10 chains, 1000 samples	0:12:10.621	0:11:19.283
10 chains, 300 samples	0:04:53.213	0:04:58.018

Table 6.15: Run times of the inference on each dataset for system *Convert* with 10 chains, using 1000 and 300 samples with 100 warmup samples.

increase the quality of results.

Revisiting systems *Encode* and *Pop* and reducing the number of samples and warmup samples generated did not produce any improvement of the results, but helped reduce the run times of the inference without loss of result quality.

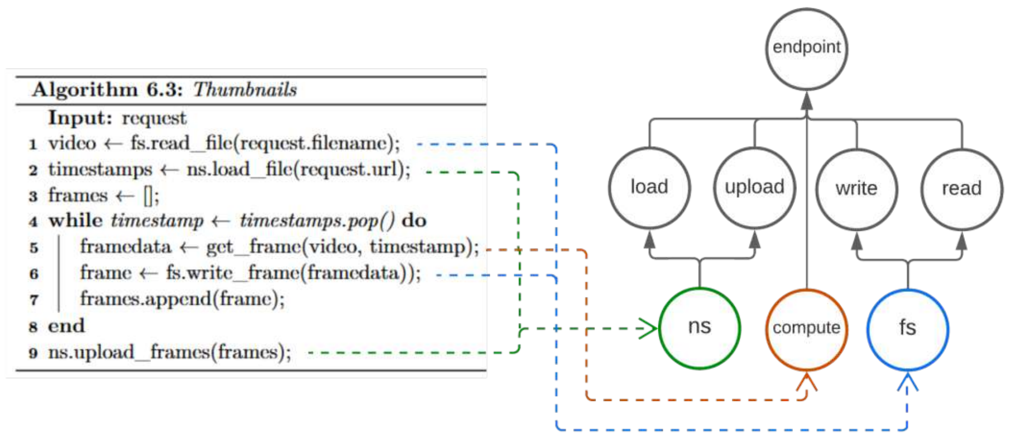
6.3 Case study: *Thumbnails*

This section is a case study evaluating the modelling process and inference with data obtained from benchmarking a real-life system. The system is a service for generating thumbnails, and to efficiently gather data while being able to manually inflict heavy-load on each component, we used the web-service described in Section 4.6, configured to represent this system and its components performing real CPU, I/O and network tasks. Similar to the data gathering of Section 4.6, we let the service run on a *Raspberry Pi Model 4* and probed the composed endpoint using the *Apache Benchmark Tool* [22]. We gathered response times reflecting performance issues by purposefully triggering heavy load on each respective component through occupying CPU cores, copying files from and to the filesystem and by transferring large files via the network and benchmarking the endpoint simultaneously.

Following the same procedure as before, we will first look at the system's code, create a performance dependency graph and model the system within the framework in Section 6.3.1 and then perform inference and evaluate the results in Section 6.3.2.

6.3.1 Definition & Modelling

Let us first look at the general behaviour of the system. Upon request to the endpoint, the system reads a video file from the filesystem and loads another file containing multiple



```
Algorithm 6.3: Thumbnails
   Input: request
1  video ← fs.read_file(request.filename);
2  timestamps ← ns.load_file(request.url);
3  frames ← [];
4  while timestamp ← timestamps.pop() do
5      framedata ← get_frame(video, timestamp);
6      frame ← fs.write_frame(framedata));
7      frames.append(frame);
8  end
9  ns.upload_frames(frames);
```

Figure 6.8: Translating the *Thumbnails* endpoint's code to a performance dependency graph.

timestamps via network, then proceeds by getting the data of each frame specified by the timestamps and finishes by writing it to an image file and uploading all files to a network storage. We can see the simplified pseudocode of this system in Algorithm 6.3.

```
Algorithm 6.3: Thumbnails
   Input: request
1  video ← fs.read_file(request.filename);
2  timestamps ← ns.load_file(request.url);
3  frames ← [];
4  while timestamp ← timestamps.pop() do
5      framedata ← get_frame(video, timestamp);
6      frame ← fs.write_frame(framedata));
7      frames.append(frame);
8  end
9  ns.upload_frames(frames);
```

Starting with Line 1, the system reads the video file using the filename given in the request, to continue on Line 2 with downloading the file containing the timestamps from the URL also specified in the request. Both lines use a respective service, `fs` for the filesystem service and `ns` for the network service. The loop on Line 4 then retrieves, for each timestamp given, the data of a single frame, invoking a CPU-heavy computation (Line 5), and writes the `framedata` to an image file on Line 6, again using the `fs` service. Finally, Line 9 concludes the process by uploading the generated frames via the network service `ns`. Lines 3 and 7 do not invoke any component exhibiting certain behaviour and contribute solely to the endpoint's overall performance.

We can again, easily translate this system to a performance dependency graph as shown in Figure 6.8. As we have previously seen with the system *Pop* in Section 6.1.2, Lines 1 and 6 warrant a separate node for the service `fs`, which is then used in nodes `read` and `write` and similarly, Lines 2 and 9 pose the need for node `ns` representing the service `ns` and being a dependency of nodes `load` and `upload`. Lastly, from Line 5 we create node `compute` to represent the computation of the `framedata` at the given timestamp.

In the same manner as before, we implement the model in *python* using the framework's templates. The resulting model can be seen in Source code 6.7. We use the templates

```python
def case_study():
    fileservice = ReadWrite('fs', lambda _: (245, 650))
    networkservice = Network('ns', lambda _: (530, 1660))
    compute = CPU('compute', lambda _: (240, 360))

    read_file = Generic('read',
                        lambda comp_times: dist.Normal(comp_times[0], 10),
                        components=[fileservice],
                        exclude=True)
    write_file = Generic('write',
                        lambda comp_times: dist.Normal(comp_times[0], 25),
                        components=[fileservice],
                        exclude=True)

    download = Generic('load',
                        lambda comp_times: dist.Normal(comp_times[0], 10),
                        components=[networkservice],
                        exclude=True)
    upload = Generic('upload',
                    lambda comp_times: dist.Normal(comp_times[0], 25),
                    components=[networkservice],
                    exclude=True)

    return Generic('case_study',
                    lambda comp_times: dist.Normal(sum(comp_times), 50),
                    components=[read_file, download, compute, upload,
    write_file],
                    exclude=True)
```

Source code 6.7: Implemented model of system *Thumbnails*

`ReadWrite`, `Network` and `CPU` for the services `fs`, `ns` and the `compute` endpoint respectively, while describing the dual usage of each of the services with a `Generic` node, sampling from a *Normal* distribution with an estimated noise around the service's performance. Finally, we compose the performance behaviour by, again, using a `Generic` node with the mean at the sum of the components' response times and a given noise as variance. The components described by `Generic` nodes are all excluded from being a possible cause for the performance issue since they all do not exhibit any impactful behaviour.

6.3.2 Inference & Evaluation

Now let us look into performing inference on the *Thumbnails* system using actual, observed data gathered as described in Section 6.3. Due to the data coming from a real system and being gathered by benchmarking this system in realtime, the size of the observed data is $n = 500$ for this inference. Thus, the initial parameters chosen for this inference are:

```
root_node = case_study()
num_samples = 500
num_chains = 3
num_warmup = 100
```
(6.4)

We present the results of this inference in the same manner as before, with the mean and standard deviation displayed in Table 6.16 and the medians displayed in Table 6.17. The run times of each dataset and parameter configuration used are shown in Table 6.18.

	`slow_chance` mean			`slow_chance` stddev		
dataset	*files.*	*networks.*	*compute*	*files.*	*networks.*	*compute*
fileservice	0.4414	0.4473	0.5548	0.2838	0.2866	0.2821
networkservice	0.5256	0.4461	0.6669	0.2893	0.2863	0.2289
compute	0.4912	0.4604	0.3899	0.3155	0.2744	0.2974

Table 6.16: Mean and standard deviation of each component's `slow_chance` of system *Thumbnails*.

	`slow_chance` median		
dataset	*fileservice*	*networkservice*	*compute*
fileservice	0.4288	0.4163	0.5844
networkservice	0.5386	0.4297	0.6998
compute	0.4864	0.4394	0.3298

Table 6.17: Median of each component's `slow_chance` of system *Thumbnails*

	run time of dataset [hh:mm:ss:SSS]		
`num_chains, num_samples`	*fileservice*	*networkservice*	*compute*
3 chains 500 samples	0:02:02.304	0:01:46.993	0:01:07.273
3 chains, 1000 samples	0:03:50.693	0:04:16.116	0:02:32.601
5 chains, 1000 samples	0:04:41.882	0:06:29.370	0:03:44.998
7 chains, 1000 samples	0:07:25.706	0:09:02.567	0:05:28.094
10 chains, 1000 samples	0:10:15.584	0:13:01.872	0:07:48.065

Table 6.18: Run times of the inferences on each dataset for system *Thumbnails*

As we can see from the results, inference with the initial parameters does not yield useful results, in fact, on the run itself, the *acceptance probability* of the different chains is varying a lot and often not hitting the default target acceptance probability of 0.8, indicating that more samples are needed during inference. Thus, we increased the `num_samples` to 1000 and `num_warmup` to 300, leading to the acceptance probability always hitting the target of at least 80%. The results for this inference run are shown in Table 6.19 and Table 6.20 respectively. Although the results for the datasets `fileservice` and `networkservice` do not differ much to the previous run, we can see an improvement in the result for dataset `compute`, with the inference actually yielding a probability of roughly 55% for component `compute` being the cause of the performance issue, as opposed to strong indifference and `compute` being the component with the lowest chance on the previous result.

	`slow_chance` mean			`slow_chance` stddev		
dataset	*files.*	*networks.*	*compute*	*files.*	*networks.*	*compute*
fileservice	0.4470	0.4434	0.5554	0.2857	0.2829	0.2828
networkservice	0.5464	0.4516	0.6554	0.2825	0.2814	0.2367
compute	0.3310	0.4412	0.5557	0.2352	0.2837	0.2826

Table 6.19: Mean and standard deviation of each component's `slow_chance` of system *Thumbnails*.

	`slow_chance` median		
dataset	*fileservice*	*networkservice*	*compute*
fileservice	0.4272	0.4227	0.5733
networkservice	0.5689	0.4277	0.7000
compute	0.2896	0.4143	0.5850

Table 6.20: Median of each component's `slow_chance` of system *Thumbnails*

Testing a further increase of `num_samples` and `num_warmup` did not provide any more improvement on the results. So the next parameter we tweaked was `num_chains`, the number of Markov chains generated. We increased the number of chains to 5, 7 and 10 while keeping the improved parameters of `num_samples=1000` and `num_warmup=300` for the inference. Using 5 chains worsened the results, with the results heavily predicting `networkservice` on every dataset. The results for inference on 7 chains start to shift away from these predictions and reather improve again, although still predicting `networkservice` wrongly on the `fileservice` and `compute` dataset. The run with 10 chains, again, yields results indicating `networkservice` as being the cause, although with less probability, but unfortunately not increasing the probability of the other components either.

Looking at the traces for the runs with 7 and 10 chains in Figure 6.9 and Figure 6.10, we can see that the `slow_chance` of `compute` and `networkservice` takes a sudden and drastic increase or decrease quite often, while the trace of `fileservice` seems to remain stable on most datasets. Similar to the result of Section 6.2.2 the sudden increase and decrease suggests that these traces might not converge at all and, in fact, testing with 15 and even 20 chains backs this assumption with the results varying back and forth and the traces showing more and more "jumps". Further, looking at the model and the defined behaviour of the components, we can see very similar response times between components in regular behaviour (see components `fs` and `compute`) and overlapping ranges of response times between components across their regular and heavy-load behaviour (see components `ns` and `fs`), which could be the reason for being unfit for inference and is a similar conclusion as drawn on one of the artificial examples, in Section 6.2.1. As a final tweak, increasing the target acceptance probability to 0.9 instead of 0.8 did not improve, but slightly worsened the results across all configurations.

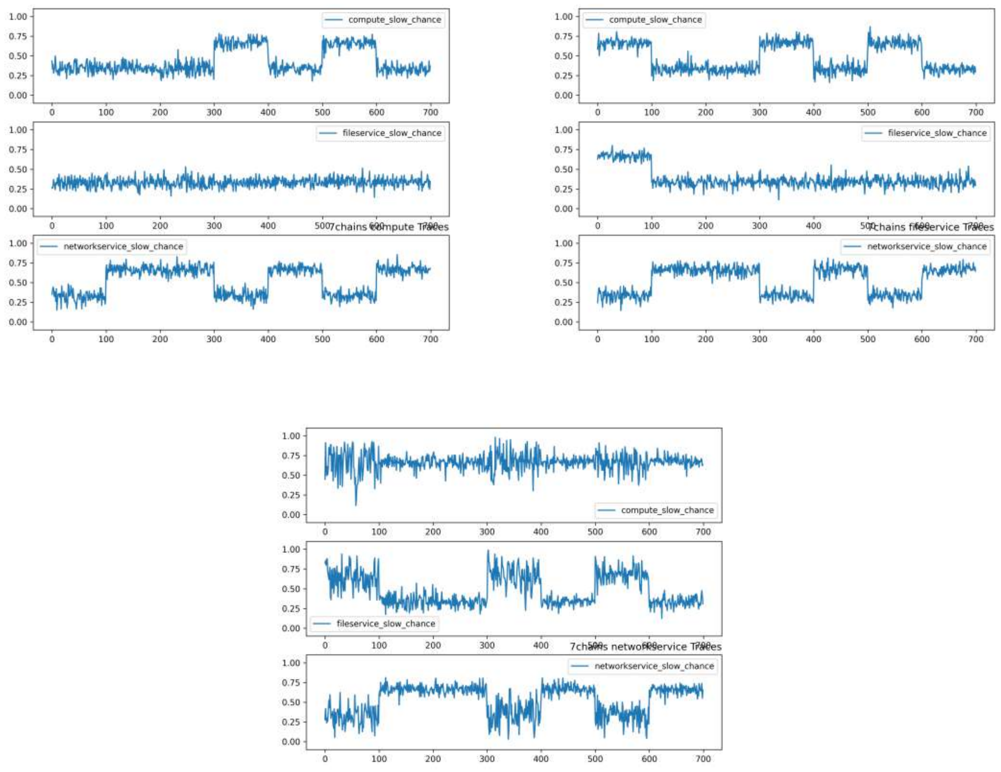


Figure 6.9: Traces of the inference on system *Thumbnails* using 7 chains.

This case study acts as a negative result, showcasing that systems as they appear and behave in real scenarios, might not be fit for reliable Bayesian inference regarding their performance. The success of such an application depends heavily on the components present in the system and the difference of the ranges of response times produced by the behaviour of these components.

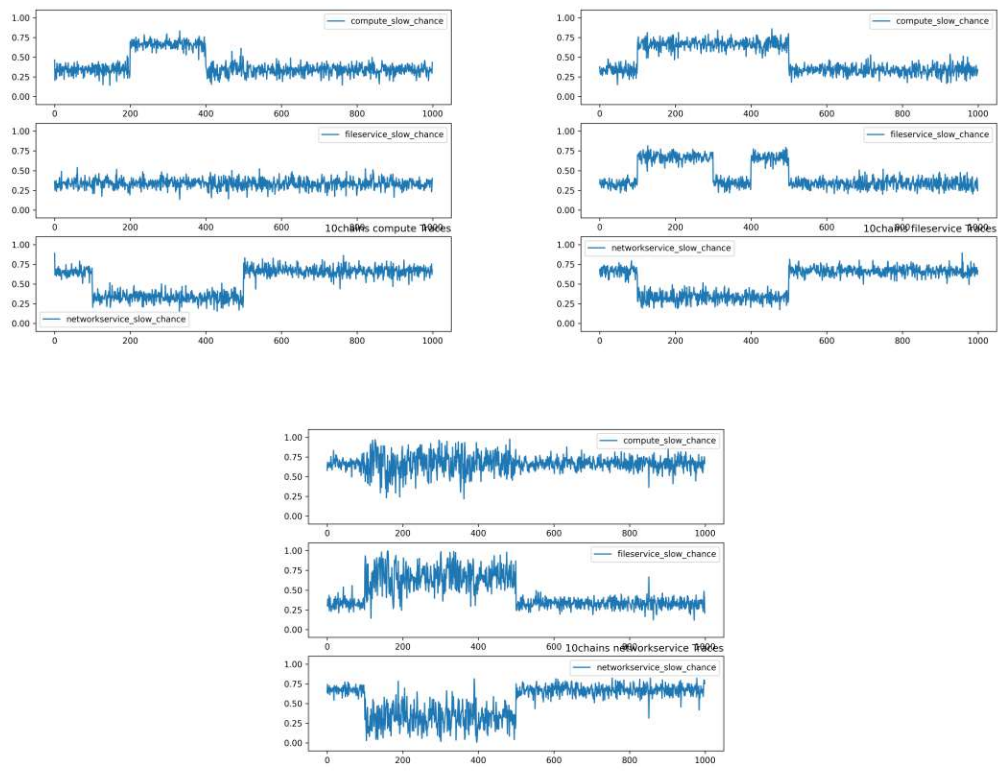


Figure 6.10: Traces of the inference on system *Thumbnails* using 10 chains.

CHAPTER 7

Conclusion

We proposed a method of modelling systems and their performance in Chapter 4 along with a way to aid in tackling inverse performance problems probabilistically through Bayesian inference and the framework described in Section 4.5. In Chapter 6 we tested this way of system modelling and evaluated inference through the framework by looking at correctness and run time of the inference, performed on three artificial systems with generated data and one actual system with gathered data.

During evaluation, the proposed method of modelling and performance representation provided a rather simple workflow of translating between code, graph and model with which, given some prior knowledge on the system's structure and behaviour, inference using the framework was quickly set up and configured. This opens the possibility of automating such a code-to-model translation with the help of, for example, static code analysis augmented by prior knowledge, although further research would be needed towards this approach. The results and information gained when running the inference on the other hand, depend highly on the system's structure and performance behaviour as well as on the number of samples and chains generated during MCMC inference. As we have seen in the evaluation of system *Encode* (Section 6.2.1), if multiple components within the system exhibit similar performance behaviour and response times, the inference of which of these components might have produced some given data becomes unreliable, with uncertainty in predictions increasing. Further, in Section 6.2.2 we could see the importance of the number of Markov chains, also observing how the quality of results decreases again with too many chains. The evaluation of system *Convert* in Section 6.2.3 showed firstly, the impact of the system's internal structure on run time, with surprisingly high run times compared to the other two systems and secondly, again the importance of tweaking the inference parameters since we could observe an increase in quality of the results when decreasing the number of samples generated, simultaneously improving run time as well. Lastly, the case study of an actual system *Thumbnails* with real, gathered data, from the system running in local network, showed that not all real systems might be fit for inference regarding their performance, especially systems with components producing similar values of response times.

In conclusion, we have shown that, given prior knowledge on computer systems and their behaviour, it is possible to rather easily translate the system and its performance into probabilistic models that can then be used in Bayesian inference, but on the other hand we could see that the inference itself needs careful configuration and in-depth testing with artificial data before being able to possibly support in solving performance problems on unseen data and even then, many systems might not even be fit for this type of inference in general. While the former observation can be used to drive research towards automation of the modelling process, the latter possibly shows the need for further refinement in the implementation of specialized inference tools such as the proposed framework and represents a major limitation of this approach.

List of Figures

List of Tables

List of Algorithms

List of source codes

Bibliography

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," *IEEE Transactions on Software Engineering*, vol. 30, p. 295–310, May 2004.

[2] J. Cao, M. Andersson, C. Nyberg, and M. Kihl, "Web server performance modeling using an m/g/1/k*ps queue," in *10th International Conference on Telecommunications, 2003. ICT 2003.*, vol. 2, p. 1501–1506 vol.2, Feb 2003.

[3] A. Mohammadi and M. R. Salehi-Rad, "Bayesian inference and prediction in an m/g/1 with optional second service," *Communications in Statistics - Simulation and Computation*, vol. 41, p. 419–435, Mar 2012.

[4] C. Sutton and M. I. Jordan, "Bayesian inference for queueing networks and modeling of internet services," *The Annals of Applied Statistics*, vol. 5, no. 1, p. 254–282, 2011.

[5] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (USA), p. 217–231, USENIX Association, 2014.

[6] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Microrca: Root cause localization of performance issues in microservices," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9, Apr 2020.

[7] T. Bayes, "Lii. an essay towards solving a problem in the doctrine of chances. by the late rev. mr. bayes, frs communicated by mr. price, in a letter to john canton, amfr s," *Philosophical transactions of the Royal Society of London*, no. 53, pp. 370–418, 1763.

[8] L. Wasserman, *Bayesian Inference*, pp. 175–192. New York, NY: Springer New York, 2004.

[9] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. A. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep universal probabilistic programming," *J. Mach. Learn. Res.*, vol. 20, pp. 28:1–28:6, 2019.

[10] T. Wiecki, “pymc-devs/pymc: v5.3.0,” *Zenodo*, 4 2023.

[11] S. D. Team, “Stan modeling language users guide and reference manual.” `https://mc-stan.org`, 2023.

[12] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with numpy,” *Nature*, vol. 585, pp. 357–362, sep 2020.

[13] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev, “Why the monte carlo method is so important today,” *WIREs Computational Statistics*, vol. 6, no. 6, p. 386–392, 2014.

[14] D. van Ravenzwaaij, P. Cassey, and S. D. Brown, “A simple introduction to markov chain monte–carlo sampling,” *Psychonomic Bulletin & Review*, vol. 25, pp. 143–154, feb 2018.

[15] F. Fischer, “Inverse performance inference.” `https://gitlab.com/flofischer/inverse-performance-inference`, 2023. Accessed: 03.05.2023.

[16] T. Chen, Y. Chen, Q. Guo, O. Temam, Y. Wu, and W. Hu, “Statistical performance comparisons of computers,” in *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, 2012.

[17] A. Khoshkbarforoushha and R. Ranjan, “Resource and performance distribution prediction for large scale analytics queries,” in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE ’16, (New York, NY, USA), p. 49–54, Association for Computing Machinery, 2016.

[18] V. Paxson, “Empirically derived analytic models of wide-area tcp connections,” *IEEE/ACM Transactions on Networking*, vol. 2, no. 4, pp. 316–336, 1994.

[19] D. G. Feitelson, *Case Studies*, p. 399–489. Cambridge University Press, 2015.

[20] D. G. Feitelson, *Statistical Distributions*, p. 73–129. Cambridge University Press, 2015.

[21] G. R. and B. R., “Weibull cumulative distribution based real-time response and performance capacity modeling of cyber–physical systems through software defined networking,” *Computer Communications*, vol. 150, pp. 235–244, 2020.

[22] T. A. S. Foundation, “ab - apache http server benchmarking tool.” `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[23] T. pandas development team, “pandas-dev/pandas: Pandas,” Feb. 2020.